Symbolic Circuit Analysis in Maple

by

Xiaofang Xie

Graduate Program
in
Applied Mathematics

Submitted in partial fulfillment
of the requirements for the degree of
Master of Science

Faculty of Graduate Studies
The University of Western Ontario
London, Ontario
Jan 2001

ABSTRACT

This thesis implements two symbolic circuit analysis methods, Mason Signal Flowgraph Analysis and Modified Nodal Analysis, in Maple. The corresponding packages are *sfg* and *mna*. *sfg* can handle circuits with all kinds of controlled sources. We also give a function *fpath* in the package *sfg*. Using *fpath*, one can find all of the paths in a graph. Using the *mna* package, a model of an operational amplifier is built. For nonlinear circuits, one can get the transfer function using *mna*.
sfg realized Mason's Signal Flowgraph algorithm with an efficient path finding algorithm. We use the compact signal flowgraph instead of a primitive signal flowgraph. This made the package more efficient. *mna* is based on modified nodal analysis, but it gets the transfer function from the element stamp of the circuit elements.

Key words: Signal Flowgraph, Modified Nodal Analysis, Mason's Formula, Path, Loop.

To my parents and my husband ...

ACKNOWLEDGEMENTS

Great thanks must go to my supervisors, Dr. Robert M. Corless and Dr. Stephen Watt. Dr. Corless got me started the topic. And I learned a lot from him, such as Maple, programming and so on. Without his continuous help, I could not finish my thesis. I also thank Dr. Watt for his encouragement.

A special thank goes to Symbolic Computation Laboratory (SCL) group. I can always get help from the people in SCL. I am lucky to be in such a group.

And also I feel thankful to my best friends here, Mr. Xiaorang Li and Ms. Xiaohe Zhang. Living with them, I always can keep a good mood.

I am pleased to have this opportunity to thank Pat for her help during my study in the Department of Applied Mathematics. Thank also goes to Mr. Usaf Aladl for his offer of the Western style file for this thesis.

Thanks could not express my thankfulness to my dear husband Long. His love and encouragement made me finish the thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Introduction

The idea of Symbolic Circuit Analysis is to keep some, or all of the circuit parameters as symbols through the entire simulation process [11]. Symbolic circuit analysis is used to generate a circuit formula which is expressed in circuit variables and complex frequency. For a long time, people paid more attention to numerical methods because of their high speed. Some mature programs [3, 32] have been developed. But in recent years, more and more people have begun to study symbolic analysis. Especially in linearized analog integrated circuits [33], symbolic analysis has been a major topic of research. The things which intrigue people are the following advantages of symbolic analysis compared to numerical methods are:

- The designer can examine the behavior of a system, rather than examine the result at some point.
- You can leave the parameters in a model as symbols.
- You can apply math operations to simplify or convert the model into related analysis techniques.

Further, good general-purpose computer algebra programs such as Maple [7, 27, 1], and Mathematica, have been developed. These programs provide convenient tools to do symbolic analysis. For example, there are some packages such as *syrup* [30] which can handle symbolic circuit analysis in Maple.

The application of symbolic analysis includes sensitivity analysis, circuit stability analysis, fault simulation, device modeling and circuit optimization [14, 13, 28, 2].

The main goal of symbolic analysis is to get a transfer function $H(x,s)$ in the following form:

$$H(x,s) = \frac{N(x,s)}{D(x,s)} \tag{1.1}$$

where x is the circuit variable, s is the complex frequency, and $N(x,s)$ and $D(x,s)$ are polynomials in x and s.

1.2 Review of Symbolic Analysis Methods

After more than thirty years' of study on symbolic analysis, there are a series of mature methods in the literature. Based on these methods, much progress has been made [20, 16, 15, 5]. According to the scheme of [9], the methods can be divided into five types:

1. Tree enumeration methods
2. Flowgraph methods
3. Parameter extraction methods
4. Interpolation methods
5. Matrix-based methods

Tree enumeration methods:

As the name of the method implies, these are used to get the transfer function by enumerating the trees of the circuit graph. The tree enumeration method has two different types: directed tree enumeration and undirected tree enumeration. Both of them create a graph from the original circuit, then enumerate the trees in the graph, and at last build the nodal admittance matrix determinant and cofactors from which one can find the transfer function. The difference is that undirected tree enumeration builds two graphs from the original circuit, one of them being the voltage graph, the other one being the current graph, while the directed tree enumeration only builds one graph from the original circuit.

The tree enumeration method can only handle circuits that consist only of resistors, inductors, capacitors or voltage controlled current sources. Our direct implemention of this method can only handle circuits in the range of 15 nodes and 30 branches.

Flowgraph methods:

These methods can also be classified into two categories: the Mason signal flowgraph and the Coates graph. The Mason signal flowgraph is based on Mason's formula. First, one constructs a signal flowgraph from the original circuit. Then, one creates Mason's formula of the circuit from the signal flowgraph. One can find the transfer function from Mason's formula. The method still has circuit size limitations if one doesn't use a hierarchical method, but it can handle circuits that contain any kind of controlled source. Some publicly available programs such as SNAP [29] and ASAP [10] are based on this method. The coates graph uses a directed graph to represent a linear system of equations. Each node of the directed graph represents one of the equations of the system. And each node (x_i) in the Coates graph represents a circuit variable and each branch weight (w_{ij}) represents a coefficient relating to x_i and x_j. For the Coates graph, the transfer function is expressed as the ratio of the graph's 1-connection, which is dependent on the input variable and the output variable selected, to the graph's 0-connection, which is independent on the input variable and the output variable selected. The Coates graph method is not used as widely as Mason's signal flowgraph methods. The idea of building a Coates graph is to use an *element stamp* method. The *element stamp* method is used to get circuit equations by inspection. The program FLOWUP [17] is based on the Coates graph method.

Parameter extraction method:

The method is used to get a transfer function by extracting symbolic parameters from the matrix formulation of the original circuit. It divides the problem into two parts. One part is in numerical form, and the other part is in symbolic form. The numerical part can be solved by standard numerical methods and then combined with the symbolic part. The rules of extracting parameters depend on the pattern of the symbolic parameters in the matrix. Different algorithms for extracting parameters have been developed to date. The parameter extraction method is good for a circuit that has few symbolic parameters. It can handle bigger circuits than the tree enumeration method and the flowgraph method. But if the number of symbolic parameters is high in the circuit, it will lose its advantage.

Interpolation methods

The method is best suited for a circuit that only has the complex frequency s as the symbolic parameter. It needs to find the coefficients of the network's determinant polynomial by evaluating it at different values of s. However it can yield incorrect solutions if the value of s is real. Therefore, a complex value of s is usually chosen. The Fast Fourier Transform is used in some implementations of the method. Obviously this is one of the advantages of this method since a Fourier transform speeds up the execution.

Matrix-based Methods:

This method builds a matrix directly from the description of the circuit. Several classical methods such as Modified Nodal Analysis (MNA), Hybrid Analysis, and Sparse Tableau Analysis can be used to generate the matrix of the circuit. After one gets the matrix of the circuit, one creates a linear equation in the following form:

$$Ax = b \tag{1.2}$$

where A is the matrix of the circuit, x is the circuit variable and b is the symbolic vector of the circuit.

The transfer function of the circuit can then be found by solving the linear system. Based on the method of solving equation (1.2), there are two categories : the determinant-based method and the parameter reduction method. The idea of the determinant-based method is to apply an extension of Cramer's rule, symbolically, to find the transfer function from the matrix in equation (1.2). Several symbolic algorithms are based on the concept of using Cramer's rule and calculating the determinant of an MNA matrix. ISAAC [12] is a program based on the determinant method. The idea of the parameter reduction methods is to apply linear algebra techniques, symbolically, to find a solution of equation (1.2). The goal here is to reduce the matrix A to two entries which are used in the transfer functions. Gaussian elimination is used in the reduction process. SCAPP [25] is a program based on the parameter reduction method.

1.3 Contribution of the Thesis

I implemented Mason Signal Flowgraph and Modified Nodal Analysis with Maple in my thesis. We can get the transfer function for a proper circuit using *sfg* and *mna*, which are packages that can easily be extended to a hierarchical method. Large-scale circuits can be handled by the hierarchical methods. Efficient algorithms for finding all paths and loops are presented in *sfg*. These algorithms can also be used in other fields related to graph theory. As an example, an operational amplifier model was built in *mna*.

1.4 Thesis Outline

Chapter 1 introduces several methods for symbolic circuit analysis. The transfer function is defined here. The advantages of symbolic analysis over numerical analysis are discussed.

Chapter 2 gives a detailed description of Mason Signal Flowgraph. The procedure for realizing the method is introduced.

Chapter 3 illustrates the precise algorithms for finding all paths and all loops.

Chapter 4 introduces Nodal Analysis and Modified Nodal Analysis. The procedure for realizing *mna* is presented.

Chapter 5 is the user guide of *sfg* and *mna*. The examples used to test the packages are given in this chapter.

Chapter 6 presents conclusions and future work.

Chapter 2

Signal Flowgraph Analysis

2.1 Introduction

"The concept of a flowgraph was originally worked out by C. E. Shannon in a classified report dealing with analog computers" [22]. In 1956 he happened to mention his work to several people at MIT who got similar results using other methods. Prior to that, his work was unknown. The greatest credit for the formulation and organization of signal flowgraph analysis is normally extended to S. J. Mason [22, 23]. In the 1950s, Mason realized the importance of flowgraphs. He worked on it, and showed how to use the signal flowgraph method to solve complicated electronic problems and provided a formula.

Because of the electronic signals and flowcharts of the systems under analysis, people gave the name *signal flowgraph analysis* to this method. The signal flowgraph method is based on Mason's rule. It is much easier to solve some problems using the signal flowgraph method than using other methods. So today the signal flowgraph method is widely used in many fields such as electronics, mechanics and statistics.

2.2 Signal Flowgraph and Mason's Rule

Before we introduce Mason's rule, we need the following basic concepts [6].

Definition 2.1 (Directed Graph) *A directed graph consists of a set of nodes and a set of branches which represented by an ordered pair of nodes in the form (x_i, x_j). This ordered pair of nodes means the branch originates from node x_i and ends at node x_j.*

Definition 2.2 (Undirected Graph) *An undirected graph here is a graph with two-way branches.*

Definition 2.3 (Path) *Imagine each directed branch in the signal flowgraph as a one-way street. A path from node x_i to node x_j is any route leaving node x_i and terminating at node x_j along which no node is encountered more than once.*

Definition 2.4 (Loop) *A loop is a path whose initial node and terminal node coincide.*

Definition 2.5 (nth-order Loop) *An nth-order loop is a set of n nontouching loops.*

Definition 2.6 (Path Weight) *The path weight is the product of all branch transmittances in a path.*

Definition 2.7 (Loop Weight) *The loop weight is the product of all branch transmittances in a loop.*

Definition 2.8 (Signal Flowgraph-SFG) *A signal flowgraph is a weighted directed graph representing a system of simultaneous linear equations according to the following three rules:*

1. *Node variables represent circuit variables (known and unknown)*
2. *Branch weights represent coefficients in the relationships among circuit variables*
3. *Every node with some incoming branches has the following equation:*

$$\begin{aligned} \textit{node variable} &= \sum(\textit{incoming branch weights} \times \textit{node variable} \\ &\quad \textit{from which the incoming branch originates}) \end{aligned}$$

where the summation is over all incoming branches (of the node under consideration).

Definition 2.9 (source Node) *In a signal flowgraph, a node with only outgoing branches is called a source node.*

Definition 2.10 (Dependent Node) *In a signal flowgraph, a node with some incoming branches is called a dependent node.*

Definition 2.11 (Sink node) *In a signal flowgraph, a dependent node with only incoming branches is called a sink node.*

Note: Electronic Circuit expresses the relationship among circuit variables. From the definition of SFG, we can see that SFG also can express this relationship, so actually, SFG can represent electronic circuit.

Now let's see the following signal flowgraphs, and from them we illustrate Mason's rule step by step.

Example 1:

Refer to Figure 2.1. The node variables are (x_0, x_1, x_2, x_3), and the branch weights are (a, b, c, d, e, f, g). According to Rule 3 in the definition of a signal flowgraph, we get the following linear equations from the SFG.

$$\begin{aligned} x_1 &= ax_0 + dx_2 \\ x_2 &= bx_0 + cx_1 + ex_2 \\ x_3 &= fx_1 + gx_2 \end{aligned} \tag{2.1}$$

That is to say, the SFG of Figure 2.1 represents the above linear equations.

In Figure 2.1, x_0 is a source node and x_1, x_2, x_3 are dependent nodes. Usually, we treat the source node as a known variable and the dependent node as an unknown variable. The transfer function is the ratio of the unknown variable to the known variable.

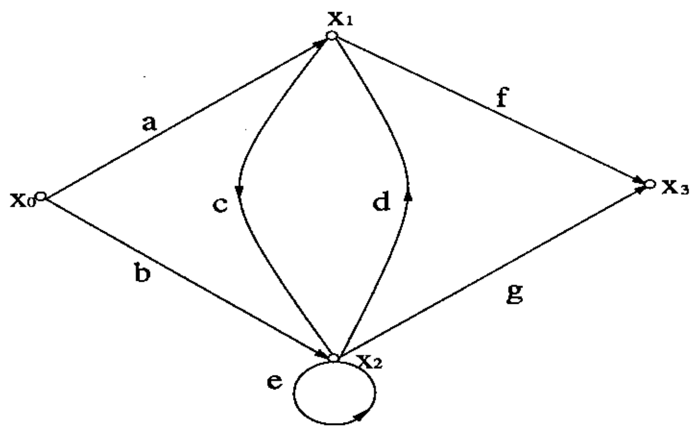


Figure 2.1: A Typical Signal Flowgraph

Example 2:

Refer to Figure 2.2

In Figure 2.2, from node x_0 to node x_4, there are three paths:

$P1 = fa, P2 = edc, P3 = gdc$

And there are two first order loops:

$L1 = ab, L2 = hd$

One second order loop:

$L1L2 = abhd$

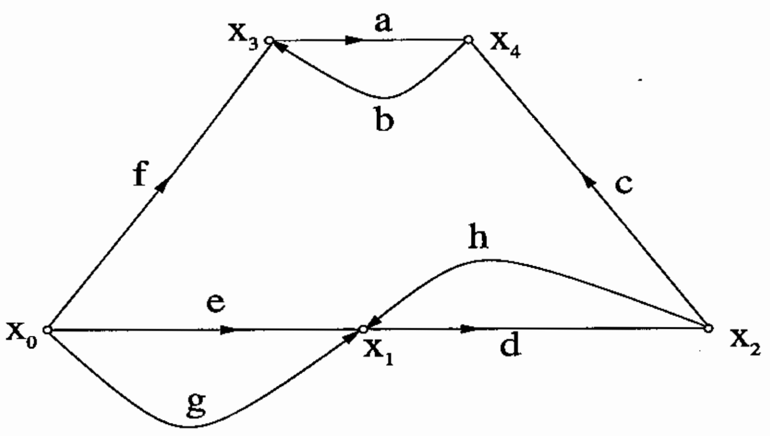


Figure 2.2: Figure For Example 2

Let's consider how to build an SFG from a system of simultaneous linear equations. Suppose the system of simultaneous linear equations can be expressed in the following form:

$$X = AX + BX_s \tag{2.2}$$

where X is the vector of unknown variables, X_s is the vector of known (source) variables and A, B are coefficients.

Then applying rule 3 from the definition of signal flowgraph, it is easy to build an SFG from the above equation. For example, $A_{i,j}$ is the weight of the branch which originates from node X_j and terminates at node X_i, $B_{i,j}$ is the weight of the branch which comes from source node X_j to dependent node X_i. In this way, we can get SFG corresponding the above equation.

Simultaneous linear equations can be solved by the use of Cramers rule, which requires the evaluation of the determinant and cofactors of the coefficient matrix. Here the determinant and cofactors can all be obtained from the signal flowgraph which is built from the simultaneous linear equations. That is why SFG provides us a direct and convenient method to solve linear equations.

Here is the idea: Solve X in equation (2.2) in terms of X_s, we get:

$X = [1 - A]^{-1}BX_s$ (when $[1 - A]^{-1}$ exists).

Then we can write X in terms of X_s as the following form:

$$X_j = T_{j1}X_{s1} + T_{j2}X_{s2} + \ldots + T_{jm}X_{sm} \tag{2.3}$$

where each T is the transfer function we want. Here, T_{ij} means transfer function from source node x_{sj} to dependent node x_i.

Mason gave the formula for T in the following form (Mason's Rule):

$$\begin{aligned} T_{ij} &\triangleq \frac{x_i}{x_{sj}} \\ &= \frac{\sum_k P_k \Delta_k}{\Delta} \end{aligned} \tag{2.4}$$

where

$$\begin{aligned} \Delta &= 1 - \text{(sum of all loop weights)} \\ &\quad + \text{(sum of all second-order loop weights)} \\ &\quad - \text{(sum of all third-order loop weights)} + \ldots \end{aligned} \tag{2.5}$$

$$\begin{aligned} P_k &= \text{weight of the } k\text{th path from the source node } x_{sj} \\ &\quad \text{to dependent node } x_i \end{aligned} \tag{2.6}$$

$$\begin{aligned} \Delta_k &= \text{sum of those terms in } \Delta \text{ without any constituent loops} \\ &\quad \text{touching } P_k \end{aligned} \tag{2.7}$$

And $\sum_k$ means all of the paths from x_{sj} to x_i.

Δ is called the graph determinant.

Let's consider the application of Mason's rule in Figure 2.2.

Example 3:

We want to calculate the transfer function of T_{40} in Figure 2.2, i. e. the ratio of x_4 to x_0.

Solution:

From example 2, we know that the paths are $P1 = fa, P2 = edc, P3 = gdc$, and that the first-order loops are $L1 = ab, L2 = hd$, and the second-order loop is $L1L2 = abhd$. Substituting the above values into equation (2.4), we get

$$\frac{x_0}{x_4} = \frac{fa(1 - hd) + edc + gdc}{1 - (ab + hd) + abhd} \tag{2.8}$$

In Mason's formula, we need to get the numerator and denominator respectively in order to get transfer function. When we make a program, we hope to get the value of the transfer function by one single process. A closed signal flowgraph provides us a convenient way to do this.

Definition 2.12 (Closed Signal Flowgraph) *A closed signal flowgaph is formed by adding a branch to the original signal flowgaph. The branch weight is $-Z$, $-Z$ is only a symbol used to distinguish it from other branch weights. And the branch originates at the output node variable and ends at the source node variable .*

Application of closed signal flowgraph

We can get the graph determinant $\triangle_c$ of a closed signal flowgraph by using equation (2.5). We can write $\triangle_c$ in the following form:

$$\triangle_c = D + CZ \tag{2.9}$$

We can show that $D = \triangle$ and $C = \sum_k P_k \triangle_k$. So we can express $\triangle_c$ in terms of $\triangle$ and $\sum_k P_k \triangle_k$:

$$\triangle_c = \triangle + Z \sum_k P_k \triangle_k \tag{2.10}$$

So now from equation (2.10) we can get the numerator and the denominator of the transfer function at the same time.

Example 4:

Suppose we want to find the transfer function of x_3/x_0 in Figure 2.1.

Solution:

First, add a branch from node x_3 to x_0 with weight $-Z$, we get Figure 2.3. In Figure 2.3, there are loops:

$$\begin{array}{lll} L_1 = e, & L_2 = cd, & L_3 = -Zaf \\ L_4 = -Zbg, & L_5 = Zacg, & L_6 = -Zbdf \end{array}$$

Second-order loop is L_1L_3

According to equation (2.5), the graph determinant of the closed graph is:

$$\begin{aligned} \triangle_c &= 1-(L_1+L_2+L_3+L_4+L_5+L_6)+L_1L_3 \\ &= (1-e-cd)+Z(af+bg+acg+bdf-eaf) \end{aligned}$$

Thus,

$$\frac{x_3}{x_0} = \frac{af-eaf+bg+acg+bdf}{1-e-cd} \tag{2.11}$$

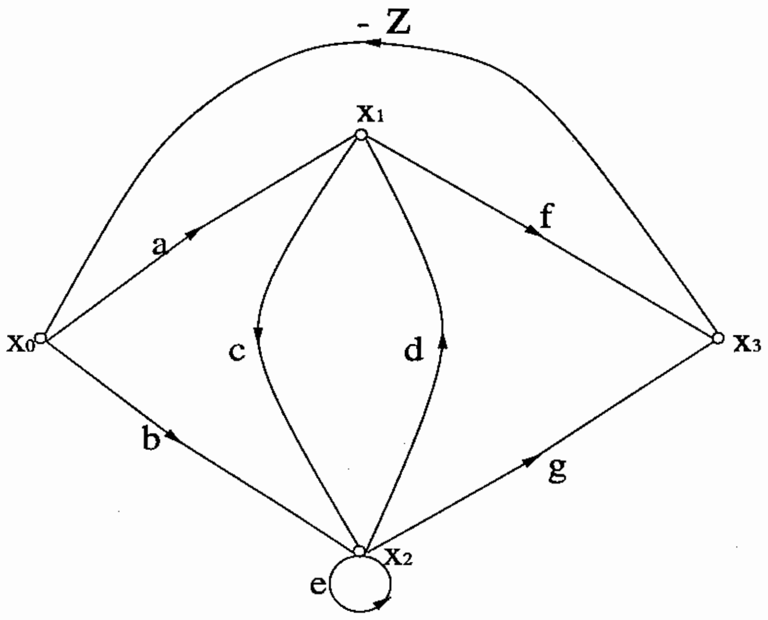


Figure 2.3: Closed SFG For Example 4

2.3 Formulation of the Signal Flowgraph

Basic Concept

Definition 2.13 (Cutset) *A cutset is a set of branches which satisfies the following conditions:*

1. *The removal of the set of branches (but not their endpoints) results in a graph that is not connected.*
2. *After the removal of a set of branches, the restoration of any one branch from the set will result in a connected graph again.*

Definition 2.14 (Current and Voltage Sources) *Current and voltage sources are active devices converting other form of energy to electrical energy. Two general categories exist for these sources in electronic circuits, independent sources and controlled sources. If the output voltage is completely independent of another voltage, then the corresponding voltage source is called an independent voltage source, otherwise, it is called a controlled voltage source. If the output current is completely independent of another current, then the corresponding current source is called an independent current source, otherwise, it called a controlled source [31, 8].*

Theorem 2.1 (KCL) *KCL (Kirchoff's Current Law) states that the algebraic sum of all current entering a node is equal to the algebraic sum of all currents leaving that node.*

Theorem 2.2 (KVL) *KVL (Kirchoff's Voltage Law) states that the sum of the voltage drops across a closed circuit is equal to 0.*

Definition 2.15 (Primitive SFG) *In the SFG, construct branches that express KCL, KVL equations and branch relationship (BR) of the circuit elements, then a Primitive SFG is formed.*

Definition 2.16 (Compact SFG) *Based on the primitive SFG, eliminate the node variables I_T and V_L, then a Compact SFG is formed.*

Now we show how to build a signal flowgraph from a circuit which contains RLC elements and independent and controlled sources.

First, we consider a circuit without controlled sources. We begin by selecting a tree and cotree for the signal flowgraph. All of the voltage sources V_E must be in the tree, and all of the current sources I_J must be in the cotree. And also we assume that the voltage sources contain no loops, the current sources contain no cutsets, and there is no self-loop in the final graph. Let the immittance tree branches be characterized by the impedance matrix Z_T, and the immittance cotree branches by the admittance matrix Y_L. According to these rules, we have the general picture of SFG in Figure 2.4. In Figure 2.4 $\odot$ represents a collection of nodes, and $\Rightarrow$ represents a collection of branches in the SFG.

Then we follow the following steps to get the primitive signal flowgraph:

1. Apply **KVL** to express each element of V_L in terms of V_E and V_T.
2. Apply **KCL** to express each element of I_T in terms of I_L and I_J.
3. For immittance tree branches, each voltage is expressed in terms of the current through the branch; i. e., $V_T = Z_T I_T$.
4. For immittance cotree branches, each current is expressed in terms of the voltage across the branch; i. e., $I_L = Y_L V_L$.

Example 5:

For the circuit in Figure 2.5, we choose branches V_a, R_d, R_e as tree branches. The rest of the branches are cotree branches. Then, according to the above steps, we get the primitive SFG shown in Figure 2.6 which displays the **KVL**, **KCL** and **BR**.

From the definition of a compact signal flowgraph, we know that the compact SFG contains less nodes and branches than the primitive SFG. So, usually, we try to find a compact signal flowgraph for the circuit. Figure 2.7 illustrates the compact signal

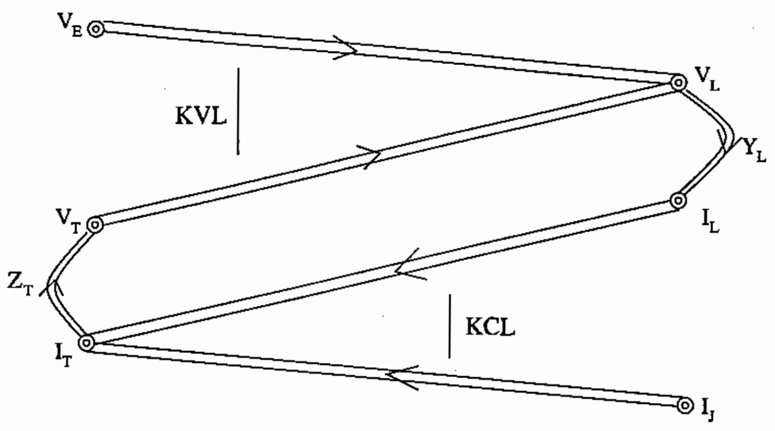


Figure 2.4: General Figure of SFG

flowgraph.

In order to get a compact signal flowgraph, we follow these steps:

1. For each immittance cotree branch Y_k, express its current in terms of tree branch voltages by the use of $I_k = Y_k V_k$ and the **KVL** equation.

2. For each immittance tree branch Z_j, express its voltage in terms of cotree branch current by the use of $V_j = Z_j I_j$ and the **KCL** equation.

Example 6:

For the circuit in Figure 2.5, we apply the above steps to get the following equations:

$$\begin{array}{lll}
I_b = G_b V_b, & V_b = V_a - V_d & \Longrightarrow I_b = G_b V_a - G_b V_d \\
I_c = G_c V_c, & V_c = V_d - V_e & \Longrightarrow I_c = G_c V_d - G_c V_e \\
V_d = R_d I_d, & I_d = I_b - I_c & \Longrightarrow V_d = R_d I_b - R_d I_c \\
V_e = R_e I_e, & I_e = I_c - I_f & \Longrightarrow V_e = R_e I_c - R_e I_f
\end{array}$$

According to the last columns in the above equations, we can build the compact signal flowgraph as Figure 2.8. It is obvious that Figure 2.8 is much simpler than Figure 2.6.

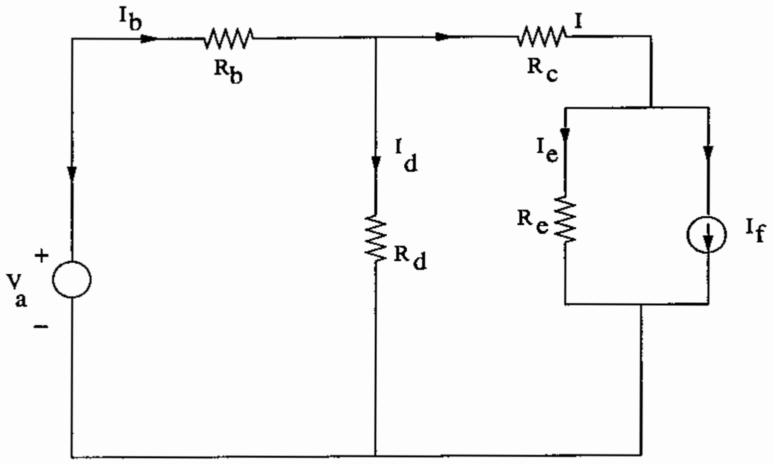


Figure 2.5: Circuit For Example 5

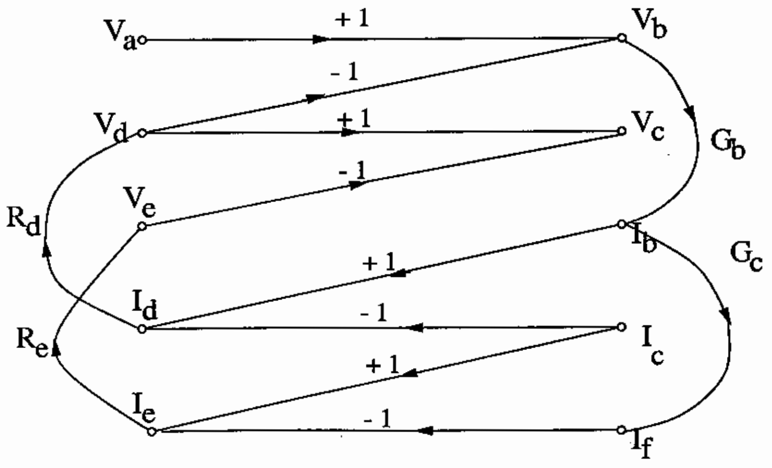


Figure 2.6: Primitive SFG For Circuit in Figure 2.5

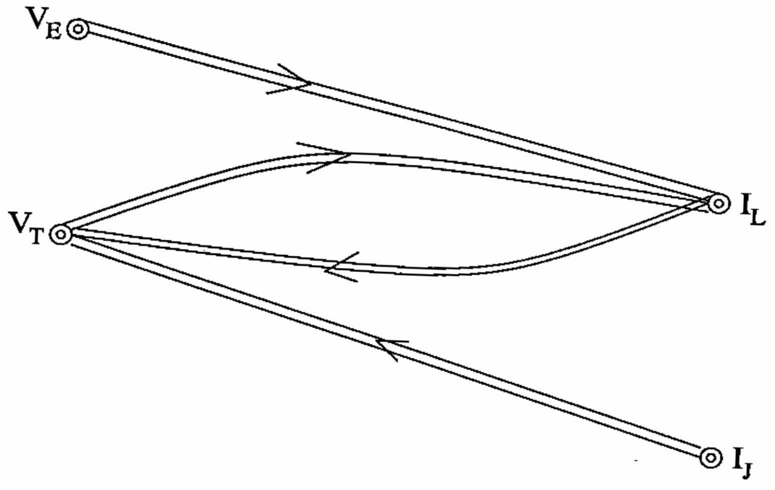


Figure 2.7: General Compact Signal Flowgraph

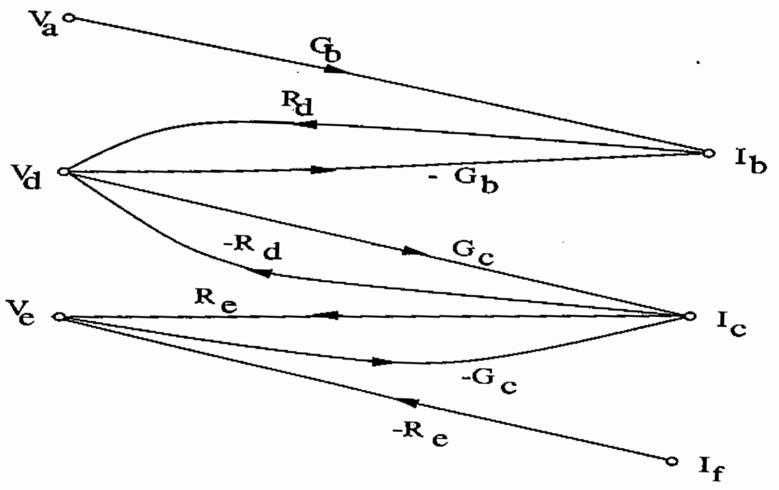


Figure 2.8: Compact SFG For Example 6

Until now we have only constructed compact SFG for circuits without controlled sources. Next we will consider compact SFG of a circuit with any type of controlled sources. Similarly, we have the assumptions: independent and controlled voltage sources form no loops, independent and controlled current sources form no cutsets. The rule for choosing the tree and cotree is: independent and controlled voltage sources must be in the tree and independent and controlled current sources must be in the cotree.

The following is a step-by-step procedure for formulating the compact SFG for linear networks containing controlled sources:

1. Temporarily replace all controlled sources by independent sources of the same type. The resultant network has no controlled sources.
2. Formulate the compact SFG for the network (without controlled sources) obtained in step 1 by the method described previously.
3. Desired outputs and all controlling variables, if not tree branch voltages and cotree currents, should be expressed in terms of the latter quantities and represented in the SFG.
4. Reinstate the constraint of all controlled sources.

Example 7:

For the circuit in Figure 2.9, we want the transfer function of V_0/V_i.

Solution:

First, we replace the controlled current source g_mV_g by an independent current source I_x. Then we get the circuit in Figure 2.10. Then we choose tree branches: V_i, R_1, R_2, R_4. Then we apply step 2 and use the method introduced previously to get the compact SFG. Expressing the current I_{R3} of the cotree branch R_3 in terms of tree branch voltage, we get the Figure 2.11. Expressing the voltages V_{R1}, V_{R2}, V_{R4} of the tree branches R_1, R_2, R_4 in terms of cotree currents, we get the Figure 2.12. Next we come to step 3, i. e. expressing the desired output voltage V_0 and the controlling voltage V_g in terms of tree branch voltages. The result is Figure 2.13. Finally, we follow step 4. Reinstate the constraint $I_x = g_m V_g$. Add a branch with weight $-Z$ from node V_0 to node V_i. The result is Figure 2.14. Now we get the compact SFG Figure 2.14 for the circuit in Figure 2.9. We can find the graph determinant of the compact SFG, then get the transfer function from it. We omit this here.

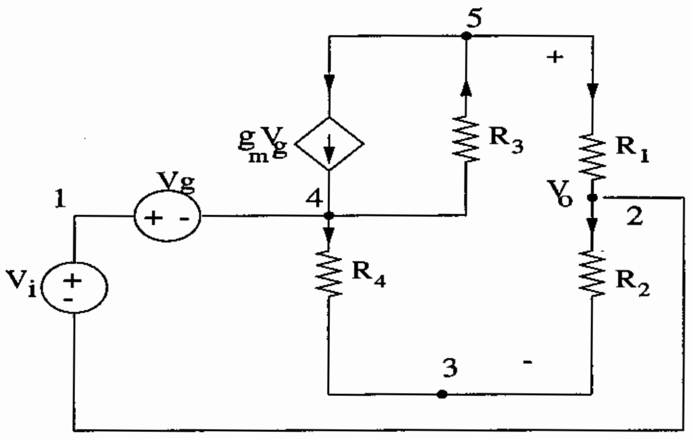


Figure 2.9: Original Circuit For Example 7

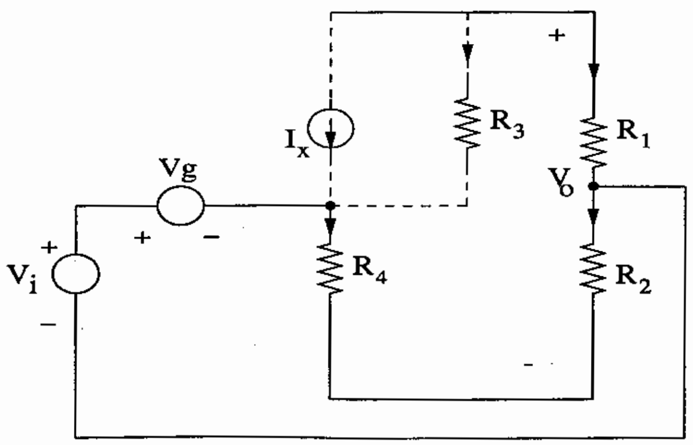


Figure 2.10: Circuit Without Controlled Source For Example 7

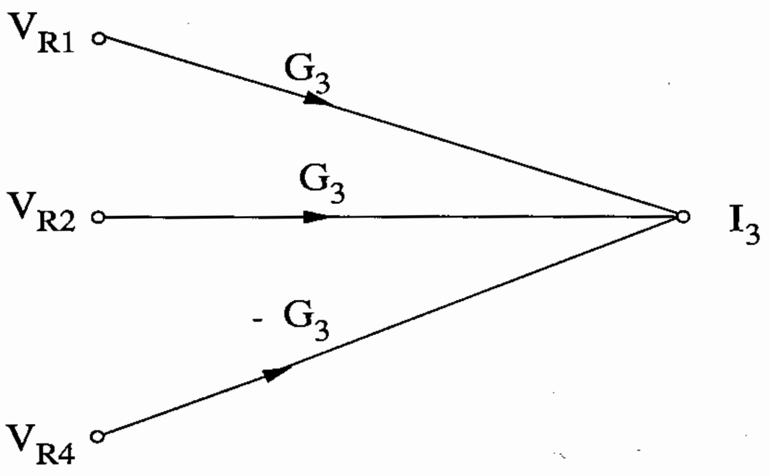


Figure 2.11: Branches Coming to I_3

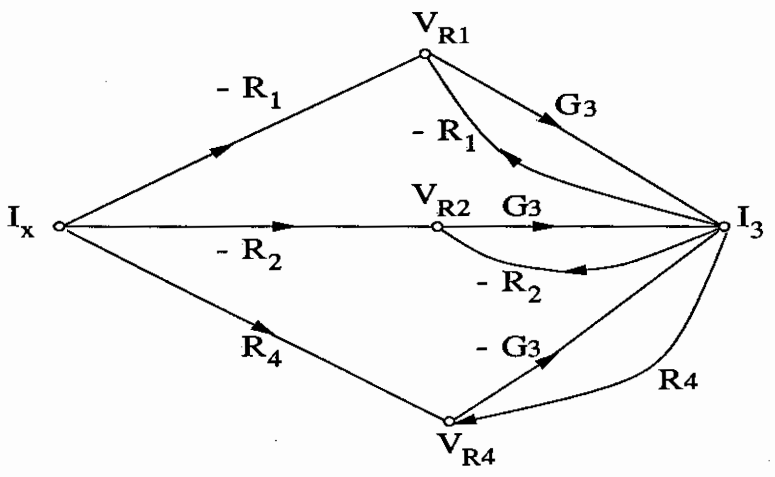


Figure 2.12: Branches Coming to V_{R1}, V_{R2} and V_{R4}

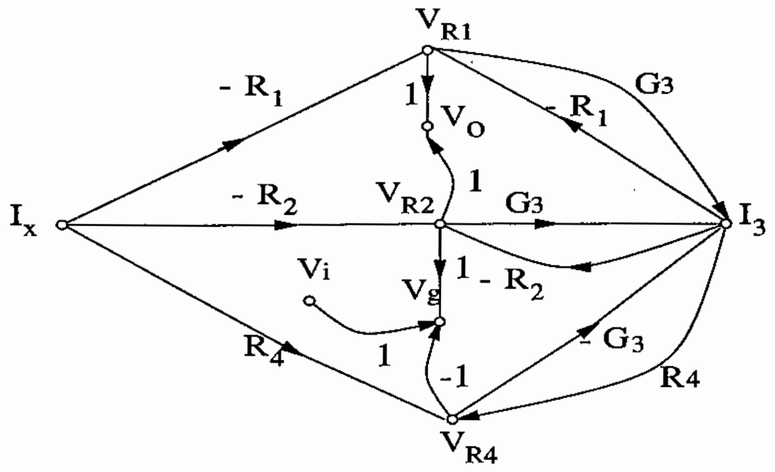


Figure 2.13: Compact SFG For Example 7

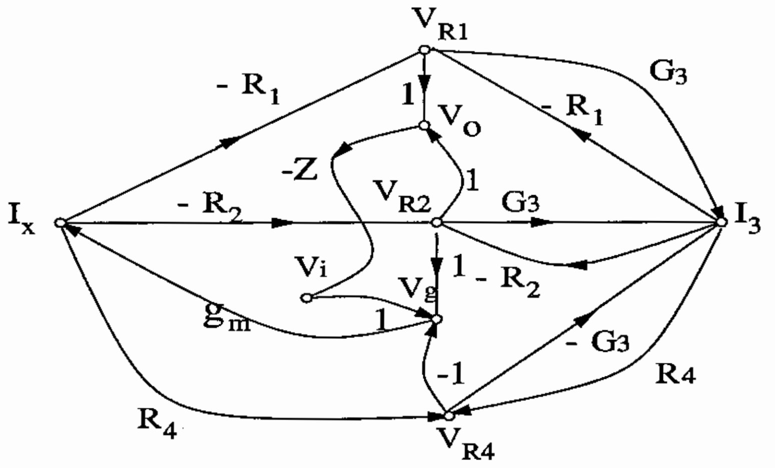


Figure 2.14: Closed Compact SFG For Example 7

2.4 Enumeration of Paths

Evaluation of Mason's formula needs to find all of the paths in the SFG. We also need to find all of the loops if we use closed signal flowgraph method. A loop is just a closed path. So the method of finding paths is important.

Imagine the directed branches of a SFG to be one-way roads and the nodes to be junctions. At each junction, one can reach different roads. Such choices can be completely described by a *routing table.*

Example 8:

For the Figure 2.15, routing table is like the table (2.1):

Note that the nodes in the second column have been listed in descending order.

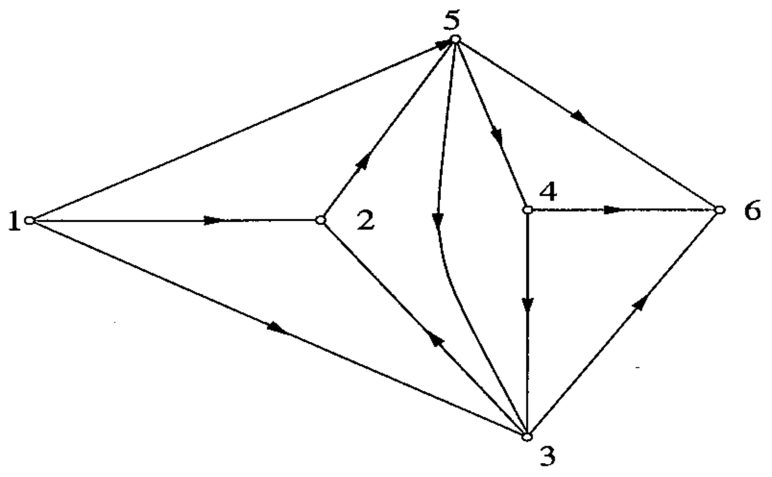


Figure 2.15: Graph For Example 8

from nodes	to nodes		
1	5	3	2
2	5		
3	6	2	
4	6	3	
5	6	4	3

Table 2.1: Routing Table for Example 8

If we want to find all paths from initial node I to the last node (some destination node) L, then we start from the initial node of the path. Referring to the routing table, proceed to the highest-numbered adjacent node. After reaching one node, use the following rules to go to the next node.

1. If the node is the destination node, a path has been found. Record the path, and apply rule 4.

2. If the node is not the destination node and had not been encountered twice on the present path from the initial node, then proceed on an untraversed branch to the highest-numbered adjacent node. If no such untraversed branch exists, apply rule 4.

3. If the node is not the destination node but has been encountered for the second time on the present route from the initial node, apply rule 4.

4. Back up one node on the present route and then proceed on an untraversed branch to the highest-numbered adjacent node. If no such untraversed branch exists, repeat the *back-up and take next choice* process. If, on backing-up, the initial node I is reached, and outgoing branches from I have been traversed, then all paths have been found.

Example 9:

We want to find all the paths from node 1 to node 6 in Figure 2.15.

Solution:

Refer to routing table (2.1). The initial node is node 1, and the last node is node 6. We begin from the first line, second column of the routing table, i. e. from node 5, proceed one node to reach node 6. Then we get the first path $P1$. Applying rule 4, back up one node to node 5 and proceed one node to node 4. Node 4 fits rule 2, so we proceed one node to node 6, and get the second path $P2$. Applying rule 4, back up one node to node 3, again, use rule 2, proceed one node to node 6, get the third path $P3$. Applying rule 4 again, back up to node 3, proceed to node 2, proceed to node 5. Note that node 5 falls into rule 3, therefore, apply rule 4, back up to node

2, repeat *back up* to node 3, node 4, node 5, then proceed to node 3, apply rule 2, proceed to node 6, get the fourth path $P4$. Continue the process until we find all of the 10 paths.

Table (2.2) displays all the paths in Figure 2.15, and Figure 2.16 illustrates the procedure of finding paths.

In the next chapter, we give the precise formulation of the corresponding algorithm.

Path	Node Sequence					
P1	1	5	6			
P2	1	5	4	6		
P3	1	5	4	3	6	
P4	1	5	3	6		
P5	1	3	6			
P6	1	3	2	5	6	
P7	1	3	2	5	4	6
P8	1	2	5	6		
P9	1	2	5	4	6	
P10	1	2	5	4	3	6

Table 2.2: All Paths of Example 9

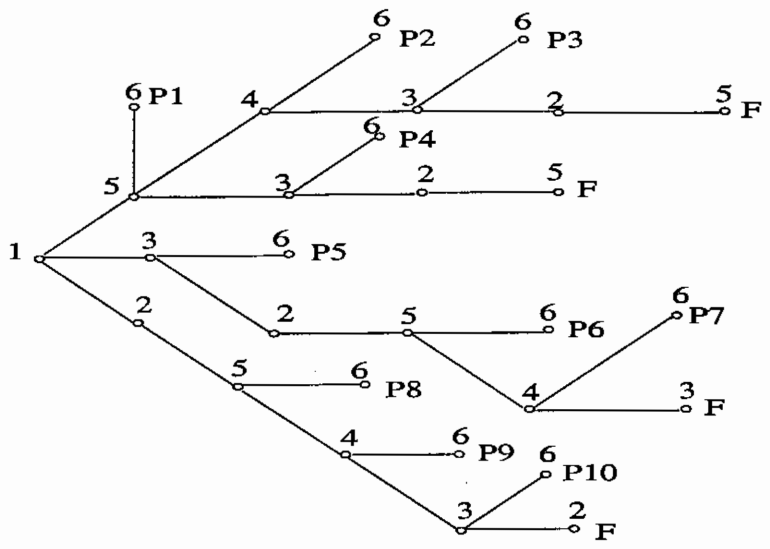


Figure 2.16: Procedure of Finding Paths

2.5 Enumeration of First-Order and nth-Order Loops

Method of Finding All First-order Loops:

First, split node i into two nodes i_1 and i_2. Node i_1 contains all incoming branches and node i_2 contains all outgoing branches. Then enumerate all paths between node i_2 and node i_1. Finally, remove all the branches which connect to node i. Repeat the procedure for node $i + 1$. After we finish finding paths for all of the nodes in SFG, then we have found all first-order loops. It is obvious that this method will generate all first-order loops without duplication.

Example 10:

Find all first-order loops in Figure 2.17.

Solution:

Applying the above method, we show in table (2.3) the ten first-order loops.

Loop	Node Sequence						
1	1	2	1				
2	4	5	4				
3	2	6	2				
4	2	6	7	8	4	3	2
5	4	8	4				
6	6	10	6				
7	6	7	8	11	10	6	
8	8	12	8				
9	9	10	9				
10	12	13	12				

Table 2.3: All First-Order Loops of Example 10

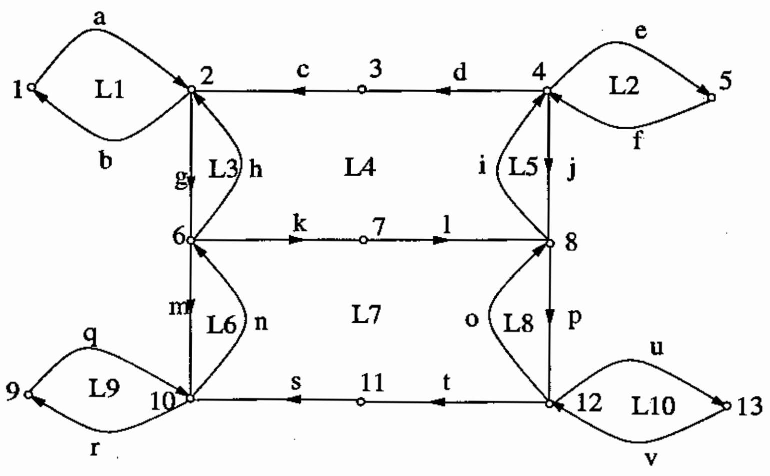


Figure 2.17: Graph For Example 10

Method of Finding All Second-order Loops:

Once we get the first-order loops, number them to 1, 2, ..., N, like table (2.3), then compare the node sequence of loop 1 with loop 2, loop 3, ..., loop N. Next, increase 1 to 2, and repeat the procedure, i.e. compare the node sequence of loop 2 with loop 3, loop 4, ..., loop N. Repeat the procedure until loop $N-1$ to get all of the second-order loops.

Example 11:

Find all second-order loops in Figure 2.17.

Solution:

Applying the above method, referring to table (2.3), we show in table (2.4) all second-order loops. Every loop in the first column together with any loop in the same row constitute a second-order loop.

Loop K	Higher-numbered loops not touching loop K						
1	2	5	6	7	8	9	10
2	3	6	7	8	9	10	
3	5	8	9	10			
4	9	10					
5	6	9	10				
6	8	10					
7	none						
8	9						
9	10						
10	none						

Table 2.4: All Second-Order Loops of Example 11

Method of Finding All nth-order Loops ($n \geq 3$):

Like the method of finding all paths, the method of finding all nth-order loops is a crucial part of any analysis program using the SFG approach. The basic idea is here: we choose loop K (initially K=1) and generate all orders of loops consisting of loops numbered K or higher. When this is done, we increase K by 1 and repeat the process. We use one example to illustrate the method:

Example 12:

Find all nth-order ($n \geq 3$) loops in Figure 2.17.

Solution:

Here we choose K=3, i. e. find all orders of loops consisting of loops numbered 3 or higher. We draw Figure 2.18 to explain the process.

The root in Figure 2.18 is numbered K, here 3. Every node (except the root and the tip nodes) in the tree graph have exactly one immediate ascendent node and some immediate descendent nodes. A node having no descendents becomes a tip node in the tree graph. The immediate descendent nodes of the root K are numerbered with those loops not touching loop K. The immediate descendent nodes of any other node corresponding to loop N are those which have the same immediate ascendent node as N and which do not touch loop N. For example, the immediate descendents of 5 are 9 and 10, 6, 9, 10 are not touching loop 5, but 6 have a different immediate ascendent with 5. Once we get the tree in Figure 2.18, we get all loops by tracing the paths from the root to all nodes in the tree. (See the result tables (2.5), (2.6), (2.7)). After performing the process successively for $K = 1, 2, \ldots, (N-1)$, we will obtain all orders of loops in the SFG.

L_1L_2	L_1L_5	L_1L_6	L_1L_7	L_1L_8	L_1L_9	L_1L_{10}
L_2L_3	L_2L_6	L_2L_7	L_2L_8	L_2L_9	L_2L_{10}	
L_3L_5	L_3L_8	L_3L_9	L_3L_{10}			
L_4L_9	L_4L_{10}					
L_5L_6	L_5L_9	L_5L_{10}				
L_6L_8	L_6L_{10}					
L_8L_9						
L_9L_{10}						

Table 2.5: All Second-Order Loops of Example 12

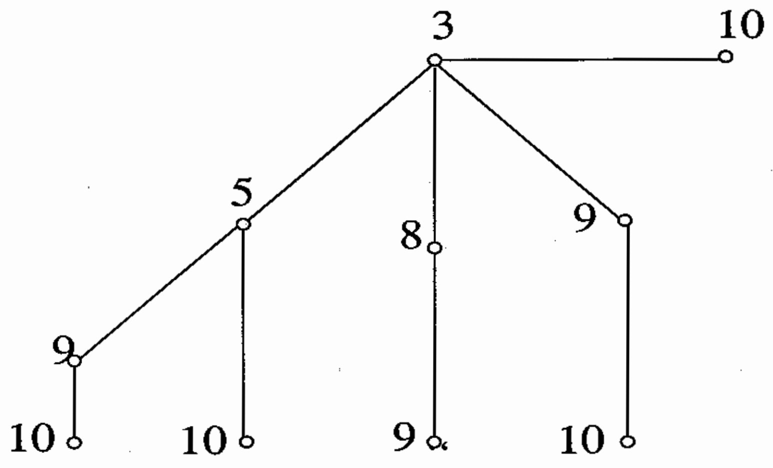


Figure 2.18: Children of Node 3 For Figure 2.17

$L_1L_2L_6$	$L_1L_2L_7$	$L_1L_2L_8$	$L_1L_2L_9$	$L_1L_2L_{10}$
$L_1L_5L_6$	$L_1L_5L_9$	$L_1L_5L_{10}$		
$L_1L_6L_8$	$L_1L_6L_{10}$			
$L_1L_8L_9$	$L_1L_9L_{10}$			
$L_2L_3L_8$	$L_2L_3L_9$	$L_2L_3L_{10}$		
$L_2L_6L_8$	$L_2L_6L_{10}$			
$L_2L_8L_9$	$L_2L_9L_{10}$			
$L_3L_5L_9$	$L_3L_5L_{10}$			
$L_3L_8L_9$	$L_3L_9L_{10}$			
$L_4L_9L_{10}$				
$L_5L_6L_8$	$L_6L_6L_{10}$			
$L_5L_9L_{10}$				

Table 2.6: All Third-Order Loops of Example 12

$L_1L_2L_6L_8$	$L_1L_2L_6L_{10}$
$L_1L_2L_8L_9$	
$L_1L_2L_9L_{10}$	
$L_1L_5L_6L_8$	$L_1L_5L_6L_{10}$
$L_1L_5L_9L_{10}$	
$L_2L_3L_8L_9$	
$L_2L_3L_9L_{10}$	
$L_3L_5L_9L_{10}$	

Table 2.7: All Fourth-Order Loops of Example 12

Chapter 3

Algorithms For Finding Paths And Finding Loops

3.1 An Algorithm for Finding Paths

Finding all paths between any two nodes of a directed graph has many applications. The success of the signal flowgraph as a tool for computer generation of symbolic network functions depends very much on an efficient path-finding algorithm. In this chapter, we will give the precise algorithm for finding all paths in a SFG. Note that there are no parallel branches in the SFG. We also assume that there is no 0 vertices in the graph.

Notation:

INI: initial path node

L: last path node

N: maximum index of node in the graph

P(w,vnode): the vth node in the sequence of path W

cj: column number of routing table

cj_list: column flag used to record the column which has been traversed

V(j,cj): routing table

counter: number of path

PF1: (preliminary)

Set $\mathtt{w} = 1, \mathtt{vnode} = 2, \mathtt{j} = \mathtt{INI}, \mathtt{P}(1,1) = \mathtt{INI}, \mathtt{N} = \mathtt{max\ index\ of\ vertices}, \mathtt{counter} =$

$0,$ `cj_list` $=$ `seq[1, i = 1...N]`

In the routing table, add node -1 as the last node in the row numbered INI and add node 0 to other rows.

PF2: (find next node)

Repeat $cj = cj\_list(j), P(w, vnode) = V(j, cj)$ for j in routing table.

PF3: (test the last node in a row of routing table)

$$signum(P(w, vnode)) = \begin{cases} 1, & PF4 \\ 0, & PF5 \\ -1, & stop \end{cases}$$

PF4: (prepare next node)

```
if $P(w, vnode)$ = last node then
  counter = counter +1;
  $j = P(w, vnode - 1)$ (back one node) ;
  $cj\_list(j) = cj\_list(j) + 1$ ;
  $P(w + 1, k) = P(w, k), k = 1 \dots vnode - 1$ ;
  $w = w + 1$;
  go to **PF2**;
if numboccur($P(w), P(w, vnode)) > 1$ ($P(w)$ is repeated ) then
  $j = P(w, vnode - 1)$ (back one node)
  $cj\_list(j) = cj\_list(j) + 1$
  go to **PF2**;
else
   $j = P(w, vnode)$
   $vnode = vnode + 1$
   go to **PF2**
```

PF5: (finish path)

$j = P(w, vnode - 1)$ (back two nodes)

$cj\_list(j) = cj\_list + 1$

```
$cj\_list(P(w, vnode-1)) = 1$
$P(w, vnode) =$ '$P(w, vnode)$' (empty P(w,vnode) for next path)
$vnode = vnode - 1$
go to PF2
```

3.2 An Algorithm for Finding Loops

Finding an nth-order loop is another important algorithm in SFG. Refer to figure 2.18 and table 2.4, rotate the figure 2.16 counter clockwise 90 degrees. If you treat each tree branch as a path, and treat the second-order loop table as the routing table in finding the paths algorithm, then the precise algorithm of finding the loops is similar to that of finding the paths. The difference is that the terminating condition is not the same.

Notation:

T(i): second-order loop table

N: number of first-order loops

P(i, w, vnode): tree branch node

cj: column number of second-order loop table

cj_list: a list used to record the column which has been traversed

j: row number of second-order loop table

counter: number of paths

parent: parent node of current node

current: current node

child: child node of current node

Algorithm:

Initialize: Let N equal the total number of the first-order loops.

```
Let $T(i)$ equal the $i$th row in the table of second-order loop, where $i$ from 1 to $N$.
Add $-1$ as the last number of $T(1)$.
Add 0 as the last number of others $T$.

for $i$ from 1 to $N$ do the following:
  let current = i, parent = i, w = 1, vnode = 1, j = i, P(i, 1, 1) = i
  and let all the value of cj_list equal 0.
  if the current root node is not 1, then
    change the last number of T(i) from 0 to $-1$
    and change the last number of T[1] from $-1$ to 0.
  if child $\neq -1$ then
    set current column number, child node, tree branch node
    if child $> 0$ then
      check if tree branch node P(i, w, vnode) is repeated in the current tree branch.
      if repeated, then
        back one node and go to the next column
        cj_list increases by 1
      if not repeated then
        check if the current node is the root node
        if yes, then
          go to the next node
        if no, then
          check if the child has the same parent with the current node
          if yes, then
            go to the next node
          if no, then
            back one node, and go to the next column.
    if child = 0 then
      back two nodes (note that parent node becomes current node),
      find new parrent node
    check if we reach tips
    if yes, then
      give the previous (vnode − 2) nodes to the next tree branch
    if no, then
      empty ($vnode$)th node of the current tree branch, i.e. empty P(i, w, vnode)
      look for ($vnode - 1$)th node for the current tree branch, i.e. look for
      P(i, w, vnode − 1)
if child = −1 then
  go to the next $i$ in for loop
```

Chapter 4

Modified Nodal Analysis

4.1 Introduction

In this chapter, we introduce how to get the transfer function by nodal analysis and modified nodal analysis. The model of operational amplifier used in *mna* is introduced also.

The key idea of modified nodal analysis is to get the nodal admittance matrix by inspection. Here, we use an *element stamp* [18] to realize it. After we get the equation of the circuit, we can use Maple to solve it directly. Thus, Maple provides us a convenient way to do symbolic circuit analysis.

4.2 Nodal Analysis

The nodal analysis [24] is based on **KCL**. In nodal analysis, we describe a circuit in the following form:

$$YV = J \tag{4.1}$$

where Y is the nodal admittance matrix, constructed by circuit parameters; V is the vector of the node voltages, and J is the vector of the independent current sources of the circuit.

J_i represents a current source entering node i. Row i of Y represents the **KCL** equation at node i. Y is created by writing the **KCL** equation at each node of the circuit. We give the following examples to illustrate nodal analysis.

Example 1:

Refer to the circuit in Figure 4.1. Write the nodal equation for the circuit.

Solution:

The nodal analysis is based on the observation that the sum of currents at any node is zero (**KCL**). At any node, there are two sets of currents, those flowing through the passive components and those due to the independent current sources. Let us write the **KCL** at each node as:

$$\sum\begin{pmatrix}\text{currents leaving the node}\\ \text{through passive components}\end{pmatrix}=\sum\begin{pmatrix}\text{currents entering the node}\\ \text{from independent sources}\end{pmatrix}$$

The above equation shows directly how the nodal admittance matrix is found by inspection. Applying this to the example, we get the following equations:

$$G_2V_1+G_3(V_1-V_2)+\tfrac{V_1-V_3}{sL_7}=J_1$$
$$G_5V_2+G_3(V_2-V_1)+sC_6(V_2-V_3)=J_4$$
$$sC_6(V_3-V_2)+\tfrac{V_3-V_1}{sL_7}+g_8V_{R3}=0$$

Because $V_{R3}=V_1-V_2$, the above equations become,

$$(G_2+G_3+\tfrac{1}{sL_7})V_1-G_3V_2-\tfrac{1}{sL_7}V_3=J_1$$
$$-G_3V_1+(G_3+G_5+sC_6)V_2-sC_6V_3=J_4$$
$$(g_8-\tfrac{1}{sL_7})V_1-(g_8+sC_6)V_2+(\tfrac{1}{sL_7}+sC_6)V_3=0$$

Writing the above equations in form of equation (4.1), we get the following equation:

$$\begin{bmatrix} G_2+G_3+\frac{1}{sL_7} & -G_3 & -\frac{1}{sL_7}\\ -G_3 & G_3+G_5+sC_6 & -sC_6\\ g_8-\frac{1}{sL_7} & -(g_8+sC_6) & \frac{1}{sL_7}+sC_6\end{bmatrix}\begin{bmatrix}V_1\\V_2\\V_3\end{bmatrix}=\begin{bmatrix}J_1\\J_4\\0\end{bmatrix}$$

Example 2:

Refer to the circuit in Figure 4.2. Write the nodal equation for the circuit.

Solution:

Writing the **KCL** for nodes 1 to 3, we get the following equations:

$$G_1V_1 + (sC_4 + G_7)(V_1 - V_2) + G_6(V_1 - V_3) = J_1 - J_6 + J_4$$
$$(sC_4 + G_7)(V_2 - V_1) + \frac{1}{sL_2}V_2 + sC_5(V_2 - V_3) = -J_4$$
$$\frac{1}{sL_3}V_3 + sC_5(V_3 - V_2) + G_6(V_3 - V_1) = J_3 + J_6$$

In form of equation (4.1), this becomes:

$$\begin{bmatrix} G_1 + sC_4 + G_6 + G_7 & -sC_4 - G_7 & -G_6 \\ -sC_4 - G_7 & \frac{1}{sL_2} + sC_4 + sC_5 + G_7 & -sC_5 \\ -G_6 & -sC_5 & \frac{1}{sL_3} + sC_5 + G_6 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ V_3 \end{bmatrix}$$
$$= \begin{bmatrix} J_1 + J_4 - J_6 \\ -J_4 \\ J_3 + J_6 \end{bmatrix}$$

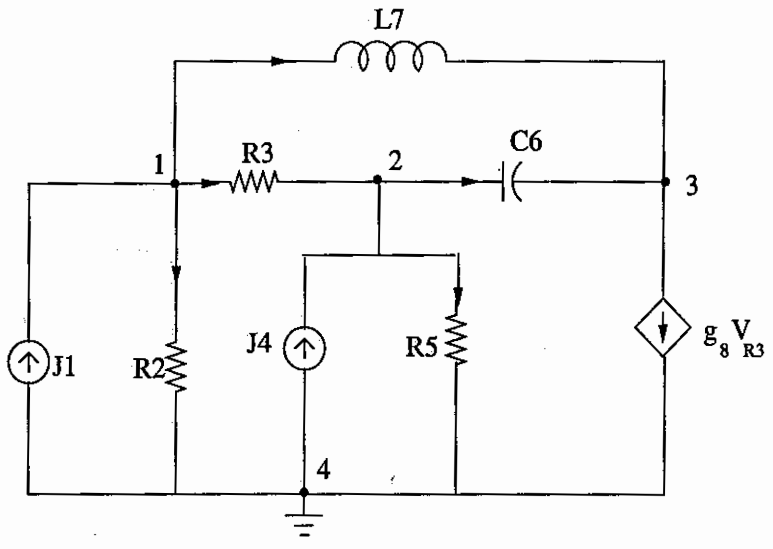


Figure 4.1: Circuit For Example 1

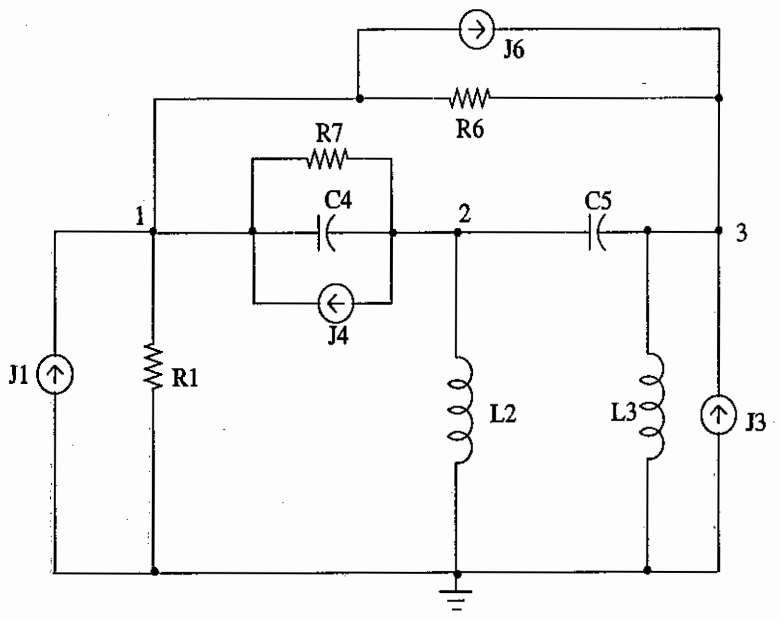


Figure 4.2: Circuit For Example 2

So we have the rules for getting the nodal equations by inspection:

1. The diagonal entries of Y are positive and

$$y_{jj} = \sum \text{admittances connected to node } j$$

2. The off-diagonal entries of Y are negative and are given by

$$y_{jk} = -\sum \text{admittances connected between node } j \text{ and } k$$

3. The jth entry of the right-hand-side vector J is

$$J_j = \sum \text{currents from the independent sources entering node } j$$

From the above rules, we deduce the *element stamp* method. It is convenient to get the equation for a circuit using *element stamp* method.

Let us consider an element with admittance connected between node i and node j, (refer to Figure 4.3). Let the current through this element be denoted by I from node i to node j. The current will appear only in the **KCL** equations associated with node i and node j, one with positive sign and the other one with negative sign:

KCL at node i: $\ldots + I \ldots = \ldots$

KCL at node j: $\ldots - I \ldots = \ldots$

where $\ldots$ indicates entries due to other elements and sources. Let us now write the current I in terms of the voltage across the element, $V_j - V_k$, and the admittance y:

KCL at node i: $\ldots + y(V_j - V_k) \ldots = \ldots$

KCL at node j: $\ldots - y(V_j - V_k) \ldots = \ldots$

Separating the terms associated with the voltages,

KCL at node i: $\ldots + yV_j - yV_k \ldots = \ldots$

KCL at node j: $\ldots - yV_j + yV_k \ldots = \ldots$

We thus see that the admittance of a two-terminal element connected between nodes i and j appears only in rows and columns i and j of Y with a positive sign at (i, i) and (j, j) and with a negative sign at (i, j) and (j, i). Refer to the *element stamp* in Figure 4.3.

For a voltage controlled current source, refer to Figure 4.4, the equation associated with $VCCS$ is :

$$I_i = g_m(V_k - V_l) \tag{4.2}$$

$$I_j = g_m(V_k - V_l) \tag{4.3}$$

Following the above procedure, we can get the *element stamp* for $VCCS$ in Figure 4.4 For an independent current source, the *element stamp* is shown in Figure 4.5.

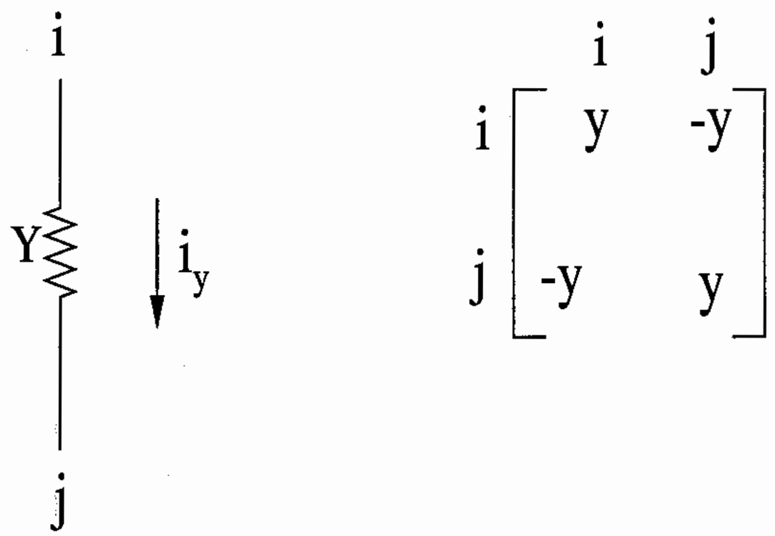
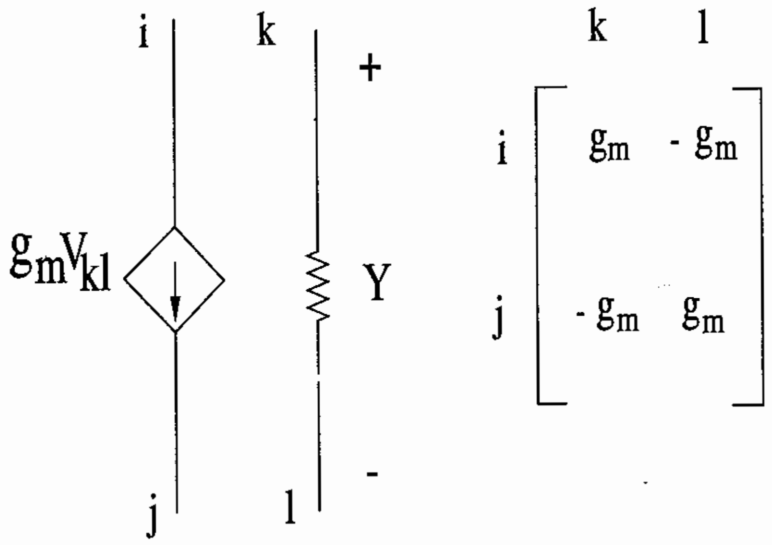
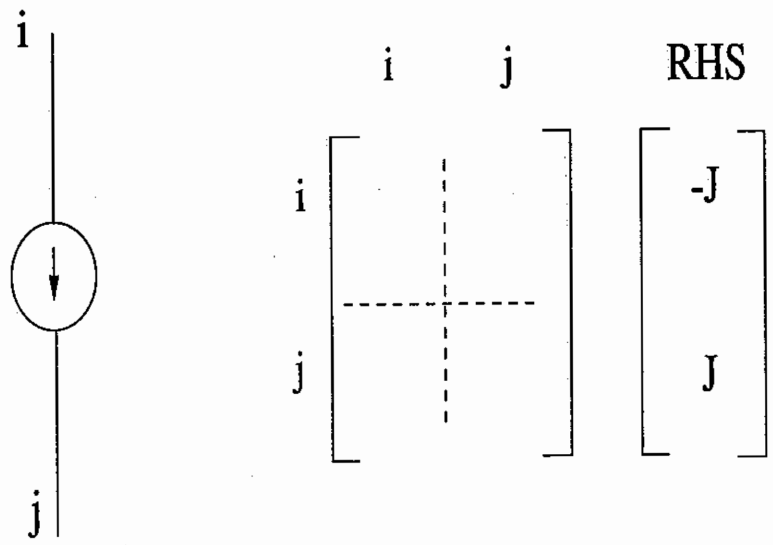


Figure 4.3: Element Stamp For R, L and C



$$\begin{array}{c|cc} & k & l \\ \hline i & g_m & -g_m \\ j & -g_m & g_m \end{array}$$

Figure 4.4: Element Stamp For VCCS



$$\begin{array}{c|cc|c} & i & j & \text{RHS} \\ \hline i & & & -J \\ j & & & J \end{array}$$

Figure 4.5: Element Stamp For Independent Current Source

4.3 Modified Nodal Analysis

Modified nodal analysis introduces independent voltage sources and three other types of controlled sources: voltage controlled voltage source ($VCVS$), current controlled current source ($CCCS$) and current controlled voltage source ($CCVS$). The key idea of modified nodal analysis is to introduce some branch currents as variables into the system of equations, which in turn allows for the introduction of any branch current as an extra system variable [9]. Each extra current variable introduced would need an extra equation to solve for it. The extra equations are obtained from the branch relationship (BR) equations for the branches whose currents are the extra variables. This results in the deletion of any contribution in the nodal admittance matrix due to the branches whose currents have been chosen as extra variables. This matrix is referred to as Y_n. The addition of extra variables and a corresponding number of equations to the system results in the the need to append extra rows and extra columns to Y_n. The new matrix Y_m is referred to as the *MNA* matrix. The new system of equations in matrix form, is

$$\begin{bmatrix} Y_n & B \\ C & D \end{bmatrix} \begin{bmatrix} V \\ I \end{bmatrix} = \begin{bmatrix} J \\ E \end{bmatrix}$$

Here I is a vector of size n_i whose elements are the extra branch current variables introduced, E is the independent voltage source values, and C and D correspond to BR equations for the branches whose currents are in I.

Using the inspection method stated above, we can find the *element stamp* for the new elements introduced by *MNA*.

1. *Independent Voltage Source:* I_v is the extra current variable and the extra equation is the BR provided by the independent voltage source, which is

$$V_i - V_j = V \tag{4.4}$$

$$I_i = I_v \tag{4.5}$$

$$I_j = -I_v \tag{4.6}$$

According to the above equations, we find the *element stamp* in Figure 4.6.

2. *Voltage Controlled Voltage Source:* I_v is the extra current variable and the extra equation is the *BR* provided by the *VCVS*, which is

$$V_i - V_j = \mu V_y \tag{4.7}$$

Substitute $V_y = V_k - V_l$ into the above equation, and adding the current equations at node i and node j, we get the following equations:

$$\begin{aligned} V_i - V_j - \mu V_k + \mu V_l &= 0 && (4.8) \\ I_i &= I_v && (4.9) \\ I_j &= -I_v && (4.10) \end{aligned}$$

According to the above equations, we find the *element stamp* in Figure 4.7.

3. *Current Controlled Voltage Source:* I_v is the extra current variable, and the extra equation is the *BR* provided by *CCVS*, which is

$$V_i - V_j = rI_y$$

Substitute $I_y = Y(V_k - V_l)$ into the above equation, and adding the current equations at node i and node j, we get the following equations:

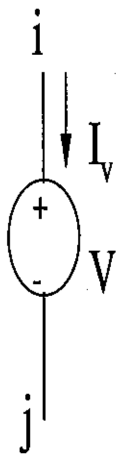
$$\begin{aligned} V_i - V_j - rV_k + rV_l &= 0 && (4.11) \\ I_i &= I_v && (4.12) \\ I_j &= -I_v && (4.13) \end{aligned}$$

According to the above equations, we find the *element stamp* in Figure 4.8.

4. *Current Controlled Current Source:* No extra variable is introduced by *CCCS*. The equation we get from *CCCS* is

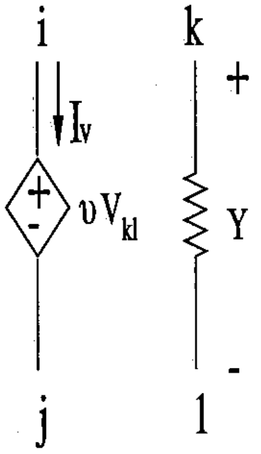
$$I_i = \beta Y(V_k - V_l) \tag{4.14}$$

We find the *element stamp* in Figure 4.9.



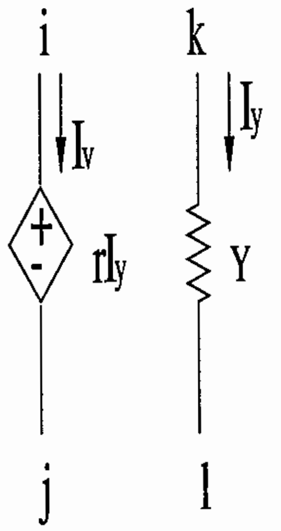
$$\begin{array}{c|cc|c|}
 & i & j & I_V \\
\hline
i & & & 1 \\
j & & & -1 \\
\hline
I_V & 1 & -1 & \\
\end{array}
\qquad
\begin{array}{c}
\text{RHS} \\
\left[\begin{array}{c} \\ \\ V \end{array}\right]
\end{array}$$

Figure 4.6: Element Stamp For Independent Voltage Source



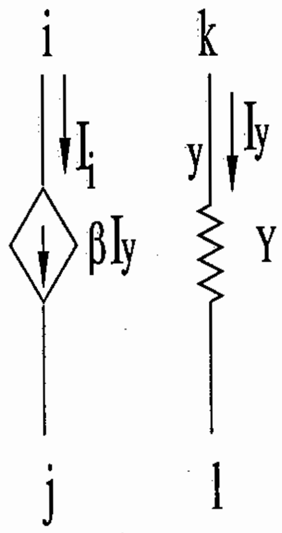
$$\begin{array}{c|cccc|c|}
 & i & j & k & l & I_V \\
\hline
i & & & & & 1 \\
j & & & & & -1 \\
\hline
I_V & 1 & -1 & -\upsilon & \upsilon & \\
\end{array}$$

Figure 4.7: Element Stamp For Voltage Controlled Voltage Source



$$\begin{array}{c|cccc|c} & i & j & k & l & I_v \\ \hline i & & & & & 1 \\ j & & & & & -1 \\ \hline I_v & 1 & -1 & -rY & rY & \end{array}$$

Figure 4.8: Element Stamp For Current Controlled Voltage Source



$$\begin{array}{c|cc} & k & l \\ \hline i & \beta Y & -\beta Y \\ j & -\beta Y & \beta Y \end{array}$$

Figure 4.9: Element Stamp For Current Controlled Current Source

Example 3:

Give the *MNA* equation for the circuit in Figure 4.10.

Solution:

The extra current variables are the branch current of the independent voltage source and the *CCVS*, they are referred to as I_1 and I_5 respectively. Using *element stamp*, we get the *MNA* equation of the circuit:

$$\begin{bmatrix} G_2 & -G_2 & 0 & 0 & 0 & 1 & 0 \\ -G_2 & G_2+G_3+G_4+\frac{1}{sL_8} & -G_4 & -\frac{1}{sL_8} & 0 & 0 & 0 \\ \beta_9 G_2 & -G_4-\beta_9 G_2 & G_4+sC_7 & -sC_7 & 0 & 0 & 1 \\ -\beta_9 G_2 & -\frac{1}{sL_8}+\beta_9 G_2 & -sC_7 & G_6+sC_7+\frac{1}{sL_8} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & G_{10} & 0 & -1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -r_5 G_3 & 1 & 0 & -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \\ V_5 \\ I_1 \\ I_5 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ J_{11} \\ 0 \\ V_1 \\ 0 \end{bmatrix}$$

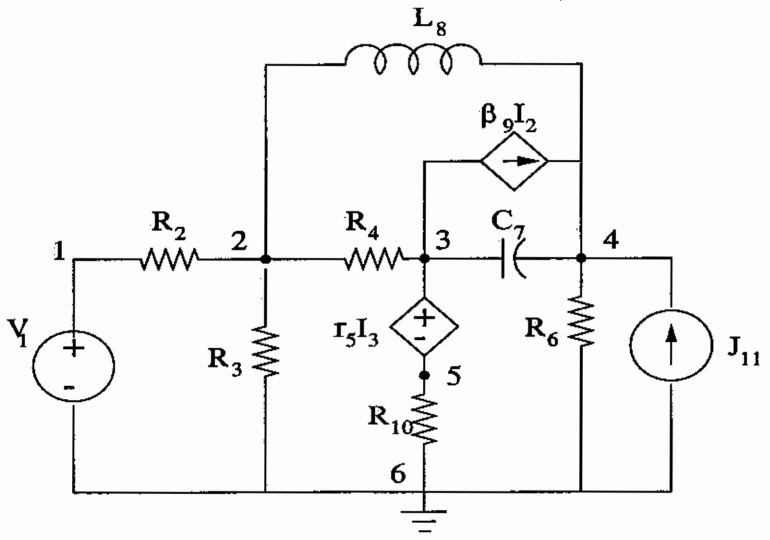


Figure 4.10: Circuit For Example 3

4.4 Operational Amplifier Model in MNA

The way operational amplifiers (opamps) are handled in *MNA* is by modeling each opamp by a voltage source with a large gain in order to attempt to characterize the infinite gain of the ideal opamp [19, 4]. The ideal opamp is a 4-terminal device for which one of the terminals is always assumed to be grounded.

Before we introduce the *element stamp* of an opamp, we give the concepts of nullator, norator and nullor [21]. They are not real elements. They are tools to introduce some mathematical constraints into a circuit. They are used as an aid to the development of insight into the behavior of ideal devices like the ideal opamp.

Definition 4.1 (nullator) *A nullator is a two terminal element defined by the constraints, which are*

$$V_i - V_j = 0 \tag{4.15}$$

$$I = 0 \tag{4.16}$$

According to the above equations, we find the *element stamp* of a nullator in Figure 4.11.

Definition 4.2 (norator) *A norator is a two terminal element for which the voltage and current are not constrained, that is*

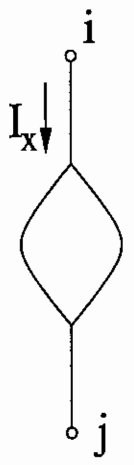
$$V_k - V_l = arbitrary \tag{4.17}$$

$$I = arbitrary \tag{4.18}$$

According to the above equations, we find the *element stamp* of a norator in Figure 4.12.

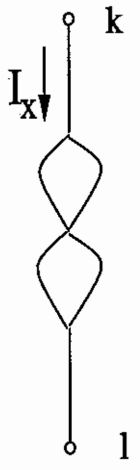
Definition 4.3 (nullor) *A nullor is produced by the combination of a nullator and a norator. Its characteristic equations are (4.15), (4.16), (4.17) and (4.18).*

According to the characteristic equations of a nullor, we get the *element stamp* of a nullor in Figure 4.13.



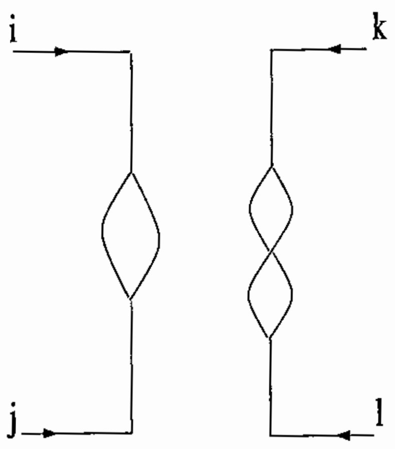
$$\begin{array}{c|cc} & i & j \\ \hline i & & \\ j & & \\ I_x & 1 & -1 \end{array}$$

Figure 4.11: Element Stamp For Nullator



$$\begin{array}{c|ccc} & k & l & I_x \\ \hline k & & & 1 \\ l & & & -1 \end{array}$$

Figure 4.12: Element Stamp For Norator



$$\begin{array}{c|ccccc}
 & i & j & k & l & I_X \\
\hline
i & & & & & \\
j & & & & & \\
k & & & & & 1 \\
l & & & & & -1 \\
I_X & 1 & -1 & & & \\
\end{array}$$

Figure 4.13: Element Stamp For Nullor

Example 4:

Write the *MNA* equation for the circuit in Figure 4.14

Solution:

The extra current variables introduced are currents of an independent voltage source and two opamps. They are referred to as I_E, I_{x1} and I_{x2} respectively. Applying the *element stamp*, we get the following equation:

$$\begin{bmatrix} G_3 & -G_3 & 0 & 0 & 0 & 1 & 0 & 0 \\ -G_3 & G_3+G_1+sC_1 & -G_1-sC_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -G_1-sC_1 & G_1+sC_1+G_2 & -G_2 & 0 & 0 & 1 & 0 \\ 0 & 0 & -G_2 & G_2+sC_2 & -sC_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & -sC_2 & sC_2+G_4 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \\ V_5 \\ I_E \\ I_{x1} \\ I_{x2} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ E \\ 0 \\ 0 \end{bmatrix}$$

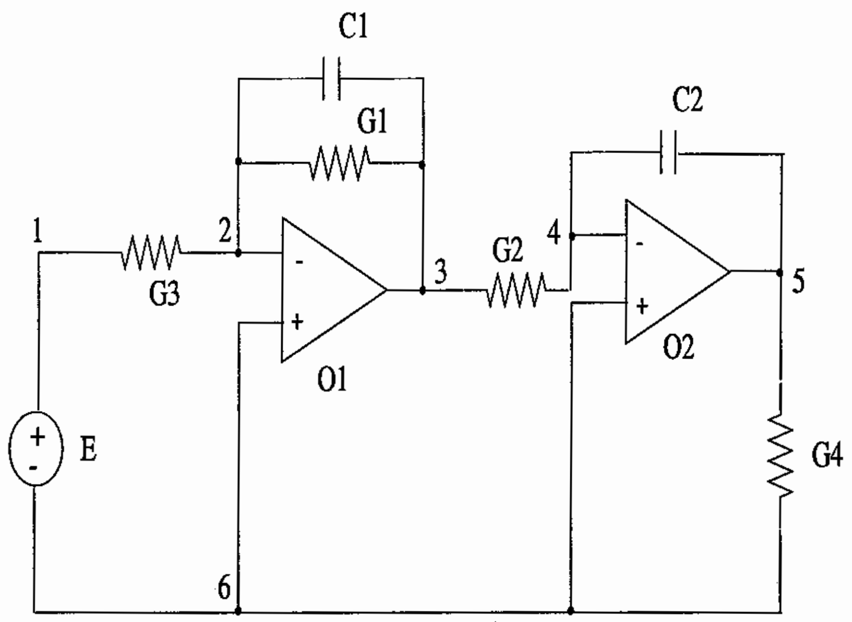


Figure 4.14: Circuit For Example 4

Chapter 5

User Guide And Examples

5.1 the sfg Package

The *sfg* package provides eight functions used to get the transfer function of a circuit. The necessary ones are *gencsfg* and *sfgmain.* The others are used to inspect the procedure in detail. We give their syntax in the format of a maple help file.

ccgraph

- Calling Sequence:

  ccgraph begininput, elementname, +node, -node, sp, element, +node, -node, . . . endinput

- Parameter

  +node and -node are the positive node and negative node of the element respectively. The inert placeholder, sp is used to separate the elements.

- Description

  ccgraph is used to input a circuit to maple. We assume that there is no parallel branch in the circuit. Different elements have different syntax:

  R, L, C: R(L, C)(string), +node, -node

  independent voltage source: V(string), +node, -node

  independent current source: J(string), +node, -node

  voltage controlled voltage source: E(string), +node, -node, controlling parameter, controlling voltage

  current controlled current source: F(string), +node, -node, controlling parameter, controlling current *voltage controlled current source:* G(string), +node, -node, controlling parameter, controlling voltage

current controlled voltage source: H(string), +node, -node, controlling parameter, controlling current

The following maple worksheet presents us some examples of *ccgraph.*

```
> with(sfg);
```

$$[ccgraph,\ ctree,\ f1loop,\ f2loop,\ fnloop,\ fpath,\ gencsfg,\ parsinp,\ sfgmain]$$

Example1:

Refer to Fig(5.1)

```
> ccgraph(begininput,V,1,2,sp,R2,3,2,sp,R1,1,3,endinput);
```

$$G,\ \{[1, 3], [3, 2], [1, 2]\}$$

```
> draw(G):
```

Refer to Fig(5.2)

Example 2:[6]

Refer to Fig(2.9)

```
> ccgraph(begininput,Vi,1,2,sp,R4,4,3,sp,R2,2,3,sp,R1,5,2,sp,
  R3,4,5,sp,G,5,4,gm,Vg,sp,Vg,1,4,endinput);
```

$$G,\ \{[4, 3], [2, 3], [5, 2], [4, 5], [5, 4], [1, 4], [1, 2]\}$$

```
> draw(G):
```

Refer to Fig(5.4)

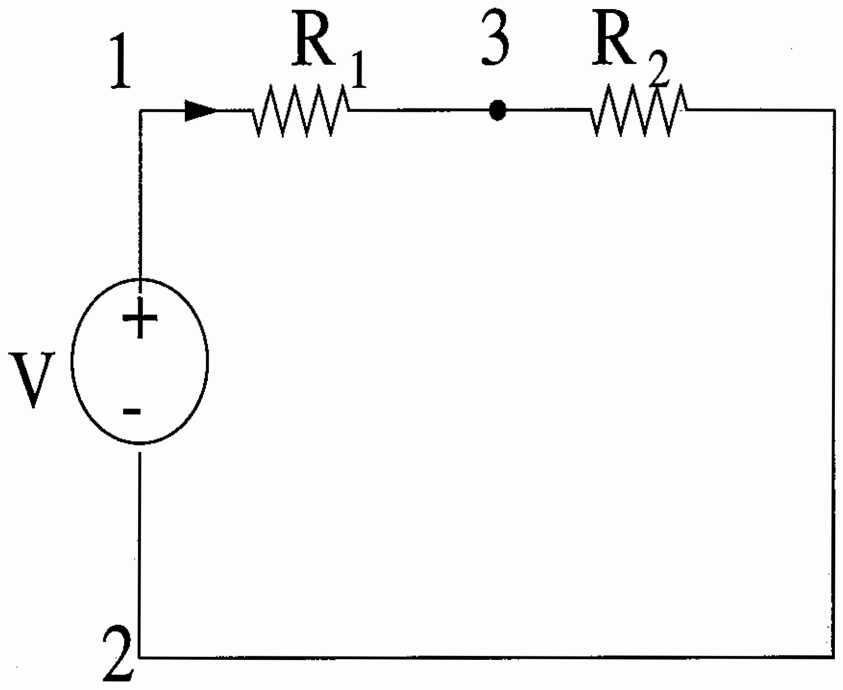


Figure 5.1: Circuit For Example 1

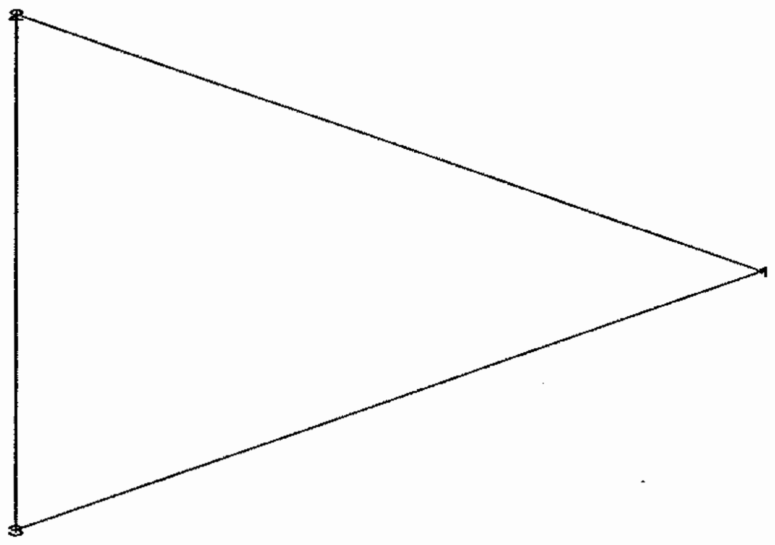


Figure 5.2: Graph Drawn by Maple For Example 1

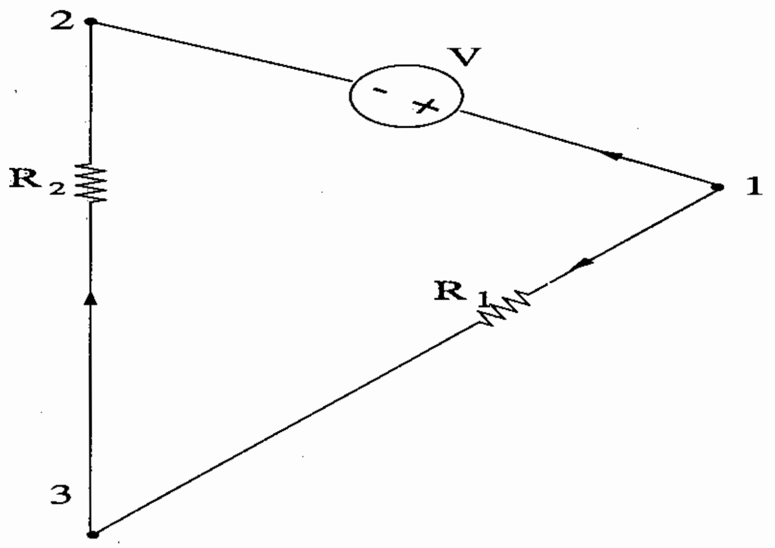


Figure 5.3: Circuit Graph Corresponding to Figure 5.2

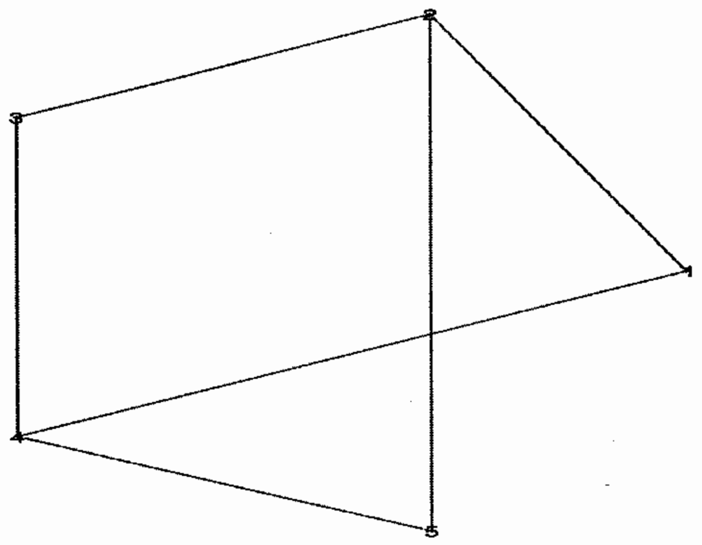


Figure 5.4: Graph Drawn by Maple For Example 2

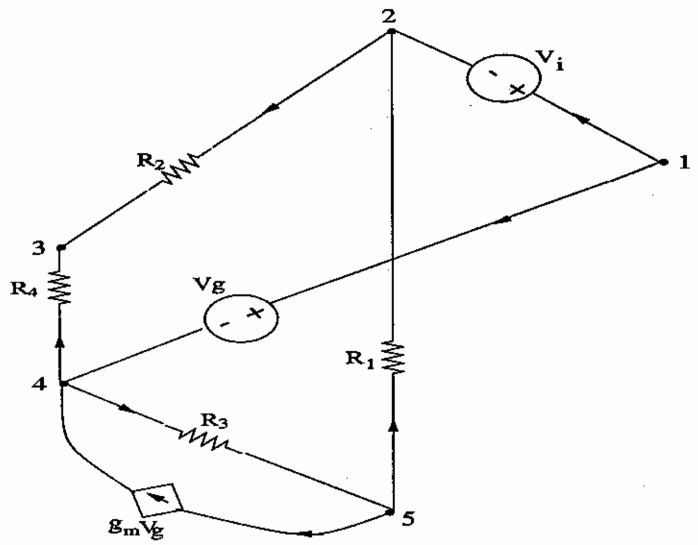


Figure 5.5: Circuit Graph Corresponding to Figure 5.4

ctree

- Calling Sequence:

  ctree G

- Parameter

  G is the graph produced by *ccgraph*

- Description

  ctree is used to produce tree and cotree from the circuit graph.

The following maple worksheet presents us some examples of *ctree*.

Example 3 :

Refer to Fig(5.1)

```
> ccgraph(begininput, V,1,2,sp,R2,3,2,sp,R1,1,3,endinput);
```

$$G,\ \{[1, 2], [3, 2], [1, 3]\}$$

```
> ctree(G);
```

$$T,\ CT$$

```
> draw(T):
```

Refer to Fig(5.6)

```
> draw(CT):
```

Refer to Fig(5.8)

Example 4:[6]

Refer to Fig(2.9)

```
> ccgraph(begininput,Vi,1,2,sp,R4,4,3,sp,R2,2,3,sp,R1,5,2,sp,
  R3,4,5,sp,G,5,4,gm, Vg,sp,Vg,1,4,endinput);
```

$$G,\ \{[1, 4], [5, 4], [4, 5], [5, 2], [2, 3], [4, 3], [1, 2]\}$$

```
> ctree(G);
```

$$T,\ CT$$

```
> draw(T):
```

Refer to Fig(5.10)

```
> draw(CT):
```

Refer to Fig(5.12)

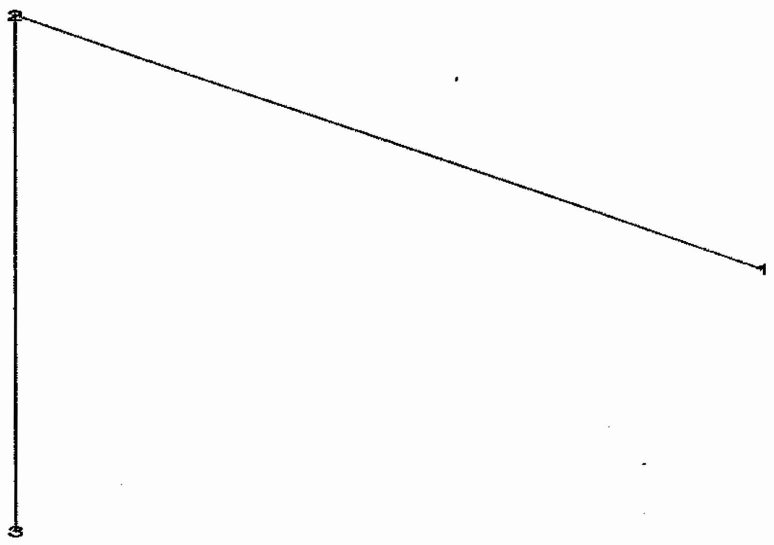


Figure 5.6: Tree Graph Drawn by Maple For Example 3

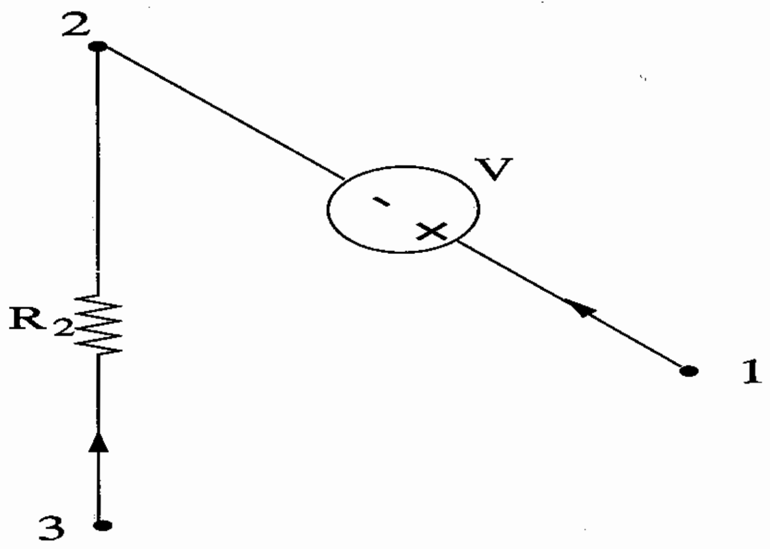


Figure 5.7: Tree Branch in Circuit Corresponding to Figure 5.6

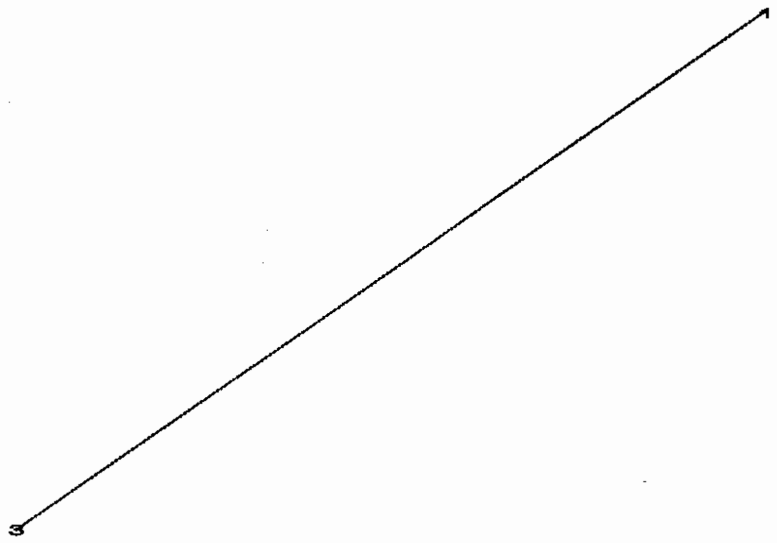


Figure 5.8: Cotree Graph Drawn by Maple For Example 3

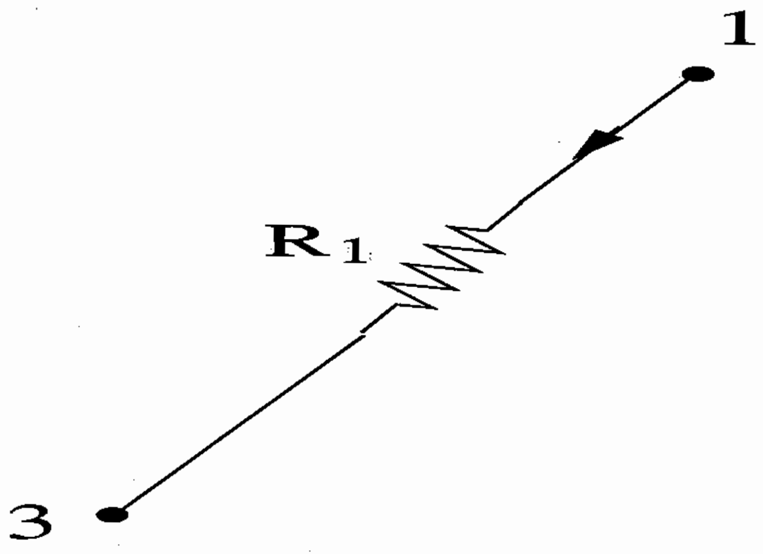


Figure 5.9: Cotree Branch in Circuit Corresponding to Figure 5.8

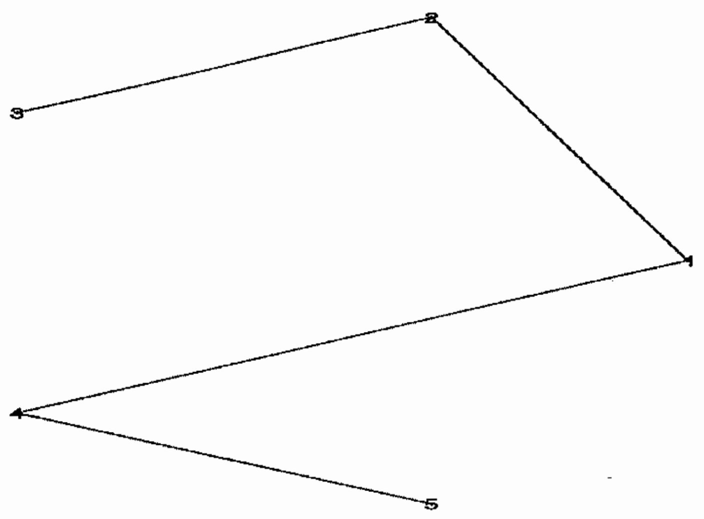


Figure 5.10: Tree Graph Drawn by Maple For Example 4

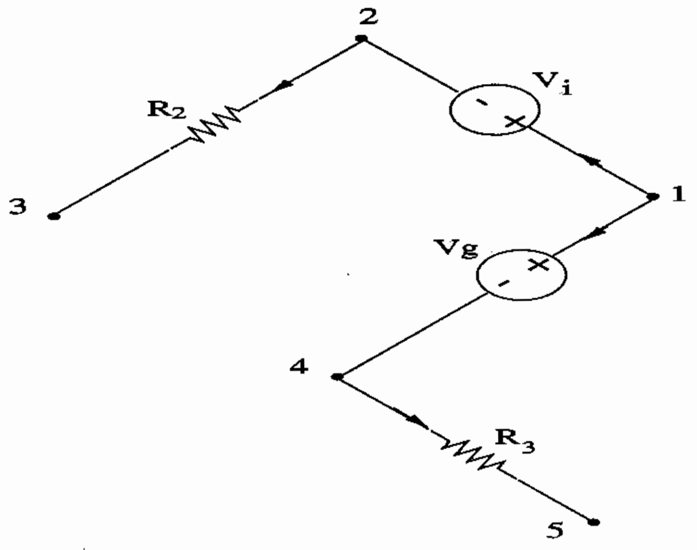


Figure 5.11: Tree Branch in Circuit Corresponding to Figure 5.10

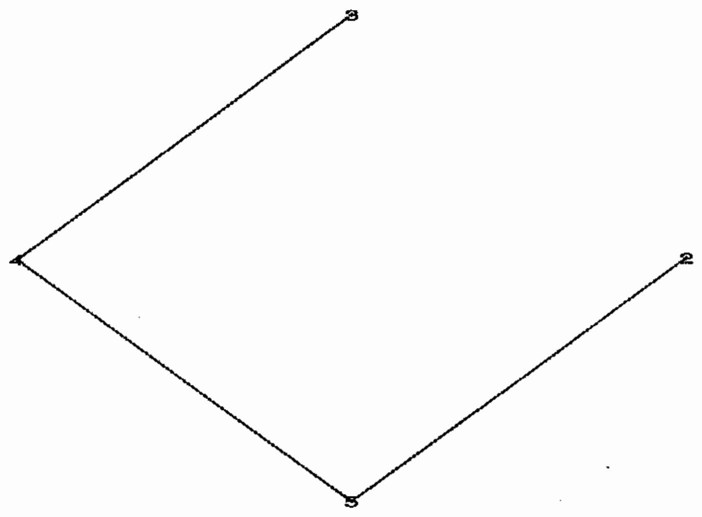


Figure 5.12: Cotree Graph Drawn by Maple For Example 4

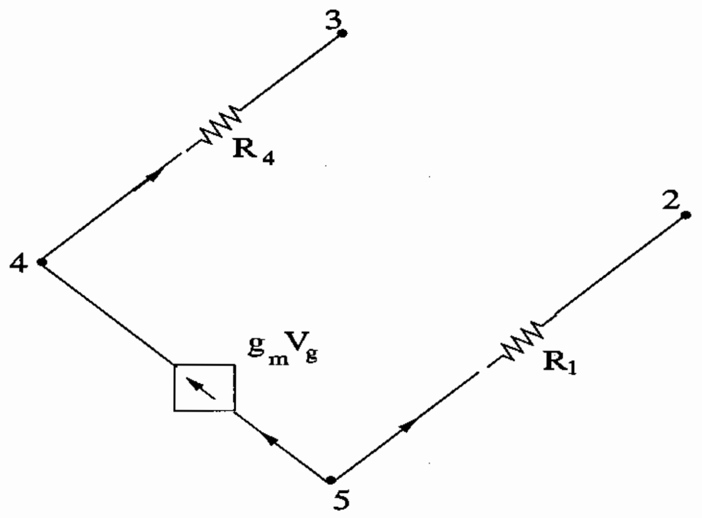


Figure 5.13: Cotree Branch in Circuit Corresponding to Figure 5.12

fpath

- Calling Sequence:

  fpath G, m, n

- Parameter

  G is the circuit graph, m is a number referring to the beginning node of the path and n is a number referring ending node of the path.

- Description

  path is used to find a path between nodes of the circuit graph. The pathcould be any type of path, directed or undirected path, in a graph. It is different from the path in Maple.

The following maple worksheet presents us some examples of *fpath*.

Example 5:[6]

Refer to Fig (5.14)

```
> G:=graph({1,2,3,4,5,6},
  {[1,2],[1,3],[1,5],[2,5],[3,6],[3,2],[4,6],[4,3],[5,6],[5,4]}):
> draw(G):
```

Refer to Fig(5.15)

```
> fpath(G,1,6);
```

9, [[1, 5, 6], [1, 5, 4, 6], [1, 5, 4, 3, 6], [1, 3, 6], [1, 3, 2, 5, 6], [1, 3, 2, 5, 4, 6], [1, 2, 5, 6], [1, 2, 5, 4, 6], [1, 2, 5, 4, 3, 6]]

Example 6:

Refer to Fig(5.16)

```
> G:=graph({1,2,3,4,5},{[1,3],[1,5],[5,2],[2,5],[2,3],[3,4],[4,2]
  }):
> draw(G):
```

Refer to Fig(5.17)

```
> fpath(G,1,4);
```

```
2, [[1, 5, 2, 3, 4], [1, 3, 4]]
```

Example 7:

```
> G:=petersen():

> draw(G):
```

Refer to Fig(5.18)

```
> fpath(G,1,10);
```

```
31, [[1, 6, 10], [1, 6, 7, 8, 9, 10], [1, 6, 7, 8, 9, 5, 4, 3, 10], [1, 6, 7, 8, 2, 3, 10],
[1, 6, 7, 8, 2, 3, 4, 5, 9, 10], [1, 6, 7, 4, 5, 9, 10], [1, 6, 7, 4, 5, 9, 8, 2, 3, 10],
[1, 6, 7, 4, 3, 10], [1, 6, 7, 4, 3, 2, 8, 9, 10], [1, 5, 9, 10], [1, 5, 9, 8, 7, 6, 10],
[1, 5, 9, 8, 7, 4, 3, 10], [1, 5, 9, 8, 2, 3, 10], [1, 5, 9, 8, 2, 3, 4, 7, 6, 10],
[1, 5, 4, 7, 8, 9, 10], [1, 5, 4, 7, 8, 2, 3, 10], [1, 5, 4, 7, 6, 10], [1, 5, 4, 3, 10],
[1, 5, 4, 3, 2, 8, 9, 10], [1, 5, 4, 3, 2, 8, 7, 6, 10], [1, 2, 8, 9, 10],
[1, 2, 8, 9, 5, 4, 7, 6, 10], [1, 2, 8, 9, 5, 4, 3, 10], [1, 2, 8, 7, 6, 10],
[1, 2, 8, 7, 4, 5, 9, 10], [1, 2, 8, 7, 4, 3, 10], [1, 2, 3, 10], [1, 2, 3, 4, 7, 8, 9, 10],
[1, 2, 3, 4, 7, 6, 10], [1, 2, 3, 4, 5, 9, 10], [1, 2, 3, 4, 5, 9, 8, 7, 6, 10]]
```

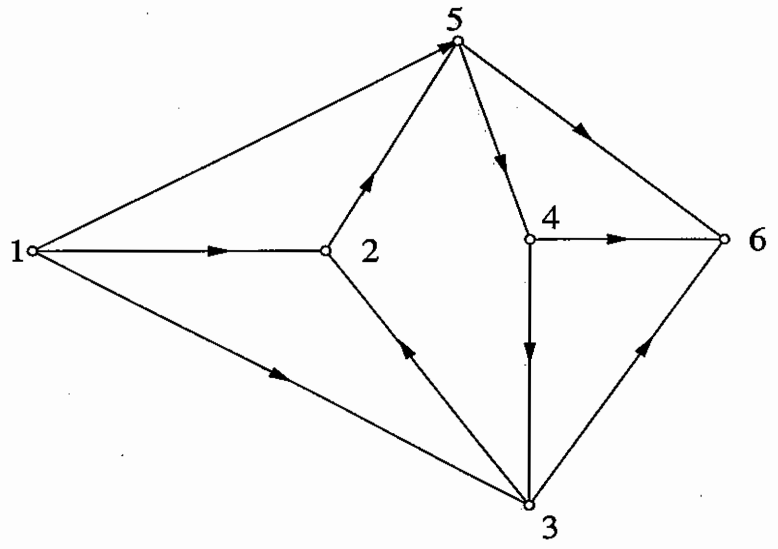


Figure 5.14: Signal Flowgraph For Example 5

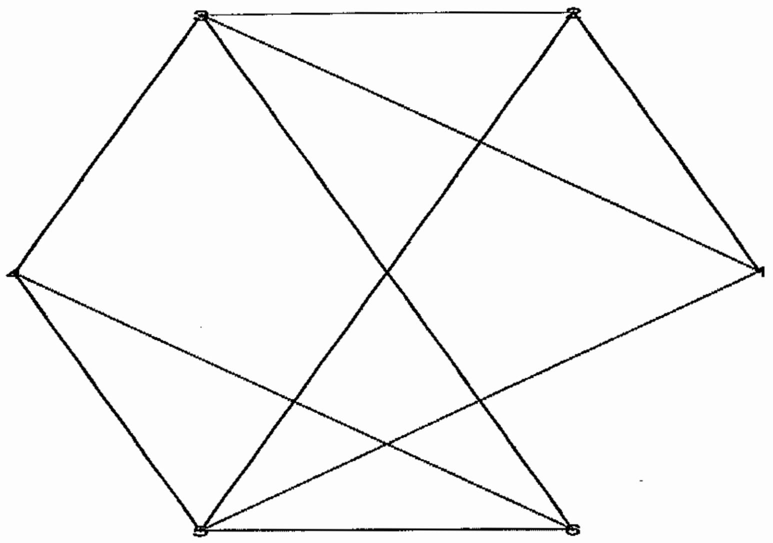


Figure 5.15: Graph Drawn by Maple For Example 5

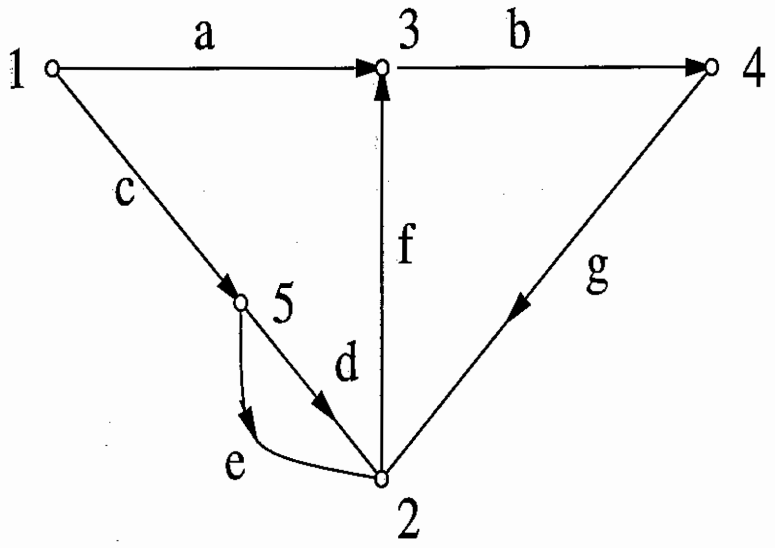


Figure 5.16: Signal Flowgraph For Example 6 and 8

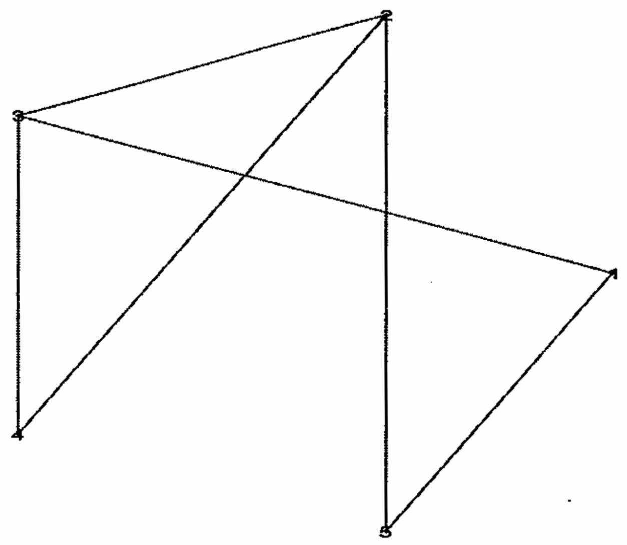


Figure 5.17: Graph Drawn by Maple For Example 6 and 8

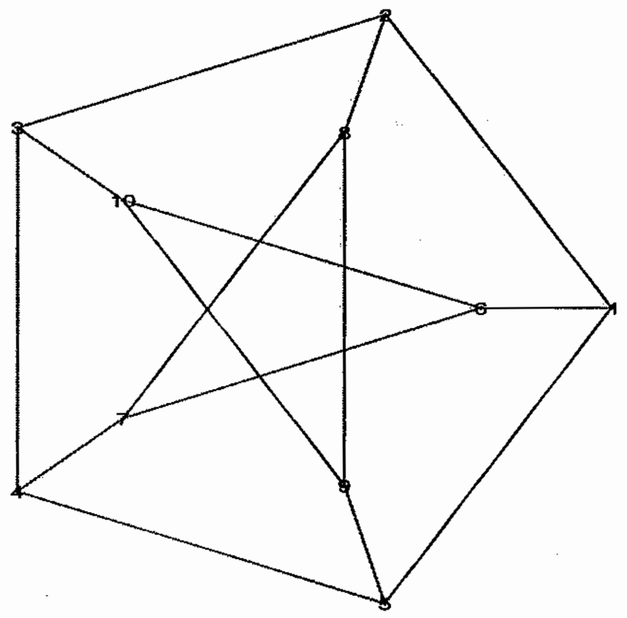


Figure 5.18: Petersen Graph Drawn by Maple For Example 7

f1loop,f2loop, fnloop

- Calling Sequence:

  f1loop G

  f2loop G

  fnloop G

- Parameter

  G is the circuit graph

- Description

  f1loop is used to find the first-order loops in the circuit graph.

  f2loop is used to find the second-order loops in the circuit graph.

  fnloop is used to find the nth-order ($n \geq 3$) loops in the circuit graph

The following maple worksheet presents us some examples of *f1loop,f2loop,fnloop*.

Example 8:

Refer to Fig(5.16)

```
> G:=graph({1,2,3,4,5},{[1,3],[1,5],[5,2],[2,5],[2,3],[3,4],[4,2]
  }):

> draw(G):
```

Refer to Fig(5.17)

```
> f1loop(G);

                    2, [[2, 5, 2], [2, 3, 4, 2]]

> f2loop(G);

                        0, [0, 0], [[], []]

> fnloop(G);

                             0, [], []
```

Example 9:

Refer to Fig(5.19)

```
> G:=graph({1,2,3,4},{[1,2],[1,3],[2,3],[3,2],[2,4],[3,4],[4,1]}):
> draw(G):
```

Refer to Fig(5.20)

```
> f1loop(G);
```

5, [[1, 3, 4, 1], [1, 3, 2, 4, 1], [1, 2, 4, 1], [1, 2, 3, 4, 1], [2, 3, 2]]

```
> f2loop(G);
```

0, [0, 0, 0, 0, 0], [[], [], [], [], []]

```
> fnloop(G);
```

0, [], []

Example 10:

Refer to Fig(5.21)

```
> G:=graph({1,2,3,4,5,6,7,8,9,10,11,12,13,14},
  {[1,11],[2,3],[2,6],[3,2],[4,5],[5,4],[5,6],[6,5],[6,2],[6,1],
  [7,2],[8,5],[9,11],[9,12],[9,10],[10,14],[10,9],[10,7],[11,9],
  [12,8],[12,13],[13,12],[14,10]}):
> draw(G):
```

Refer to Fig(5.22)

```
> f1loop(G);
```

10, [[1, 11, 9, 12, 8, 5, 6, 1], [1, 11, 9, 10, 7, 2, 6, 1], [2, 6, 2], [2, 3, 2], [4, 5, 4], [5, 6, 5], [9, 11, 9], [9, 10, 9], [10, 14, 10], [12, 13, 12]]

```
> f2loop(G);
```

27, [2, 2, 5, 6, 4, 4, 2, 1, 1, 0], [[4, 9], [5, 10], [5, 7, 8, 9, 10], [5, 6, 7, 8, 9, 10], [7, 8, 9, 10], [7, 8, 9, 10], [9, 10], [10], [10], []]

```
> fnloop(G);
```

```
42, [1, 2, 10, 15, 5, 5, 2, 1, 1, 0], [[[1, 4, 9, 10, 0]], [[2, 5, 10, 0], [2, 10, 0]], [
[3, 5, 7, 9, 10, 0], [3, 5, 7, 10, 0], [3, 5, 8, 10, 0], [3, 5, 9, 10, 0], [3, 5, 10, 0],
[3, 7, 9, 10, 0], [3, 7, 10, 0], [3, 8, 10, 0], [3, 9, 10, 0], [3, 10, 0]], [
[4, 5, 7, 9, 10, 0], [4, 5, 7, 10, 0], [4, 5, 8, 10, 0], [4, 5, 9, 10, 0], [4, 5, 10, 0],
[4, 6, 7, 9, 10, 0], [4, 6, 7, 10, 0], [4, 6, 8, 10, 0], [4, 6, 9, 10, 0], [4, 6, 10, 0],
[4, 7, 9, 10, 0], [4, 7, 10, 0], [4, 8, 10, 0], [4, 9, 10, 0], [4, 10, 0]],
[[5, 7, 9, 10, 0], [5, 7, 10, 0], [5, 8, 10, 0], [5, 9, 10, 0], [5, 10, 0]],
[[6, 7, 9, 10, 0], [6, 7, 10, 0], [6, 8, 10, 0], [6, 9, 10, 0], [6, 10, 0]],
[[7, 9, 10, 0], [7, 10, 0]], [[8, 10, 0]], [[9, 10, 0]], [0]]
```

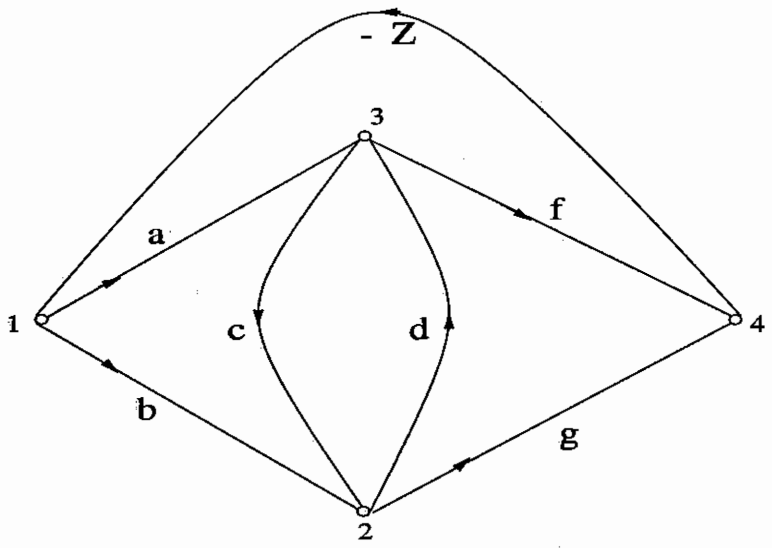


Figure 5.19: Signal Flowgraph For Example 9

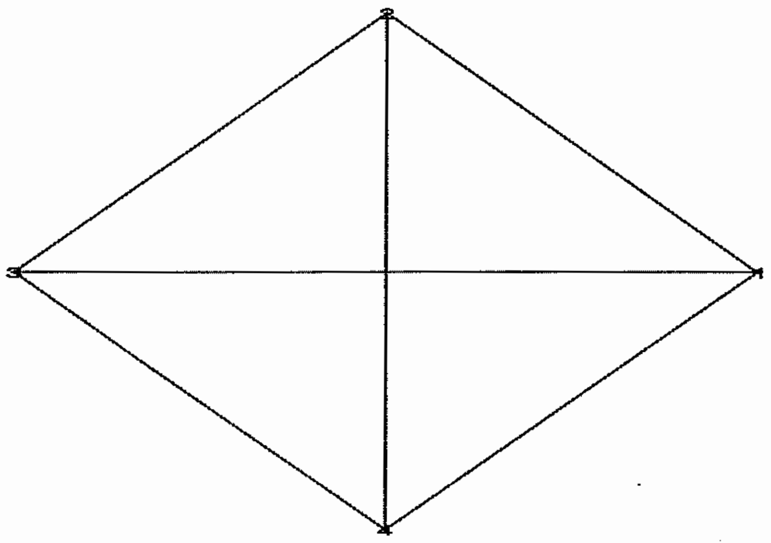


Figure 5.20: Graph Drawn by Maple For Example 9

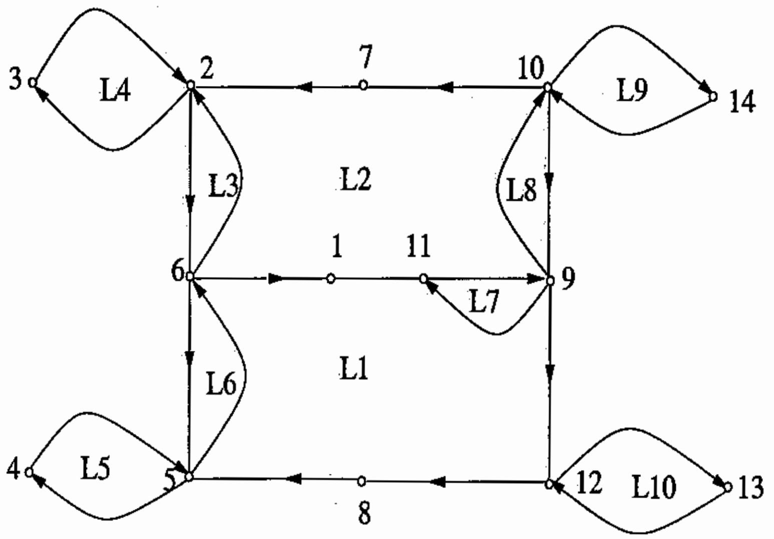


Figure 5.21: Signal Flowgraph For Example 10

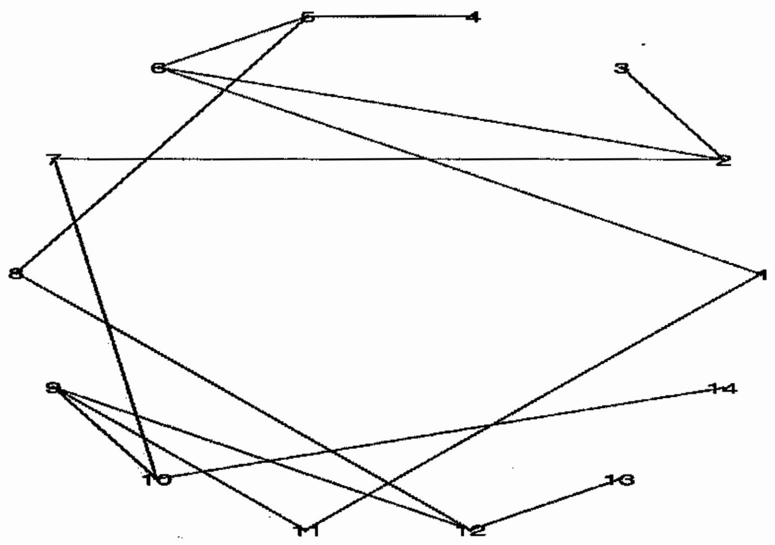


Figure 5.22: Graph Drawn by Maple For Example 10

gencsfg

- Calling Sequence:

  gencsfg begininput, element, +node , -node, sp, element, +node, -node, ..., endinput

- Parameter

  +node and -node are the positive node and the negative node of the element respectively. sp is used to separate the elements.

- Description

  gencsfg is used to generate the compact signal flowgraph of the circuit . But the vertice of the graph produced is symbolic. So we need to code it in *parsinp*.

sfgmain

- Calling Sequence:

  sfgmain G, outvar, invar

- Parameter

  G is the graph produced by *gencsfg*, outvar is the output variable in the transfer function, invar is the input variable in the transfer function.

- Description

  sfgmain is used to get the transfer function for the circuit.

The following maple worksheet presents us some examples of *sfgmain*.

Example 11:

Refer to Fig(5.1)

```
> gencsfg(begininput, V,1,2,sp, R2,3,2,sp,R1,1,3,endinput);
```

$$RTG$$

```
> sfgmain(RTG,VV,VR1);
```

$$\frac{R1}{R2 + R1}$$

```
> sfgmain(RTG,VV,VR2);
```

$$\frac{R2}{R2+R1}$$

Example 12:

Refer to Fig(5.23)

```
> gencsfg(begininput, V,1,2,sp,R1,1,3,sp,R2,3,2,sp,R3,2,1,endinput);
```

$$RTG$$

```
> sfgmain(RTG,VV,VR1);
```

$$\frac{R1}{R2+R1}$$

```
> sfgmain(RTG,VV,VR2);
```

$$\frac{R2}{R2+R1}$$

```
> sfgmain(RTG,VV,VR3);
```

$$-1$$

Example 13:

Refer to Fig(5.24)

```
> gencsfg(begininput, V,1,2,sp,R,3,2,sp,C,4,3,sp,L,1,4,endinput);
```

$$RTG$$

```
> sfgmain(RTG,VV,VR);
```

$$\frac{RSC}{1+S^2LC+RSC}$$

Example 14:

Refer to Fig(5.25)

```
> gencsfg(begininput,V,1,3,sp,R,3,2,sp,C,1,2,sp,G,2,3,gm,V,endinput);
```

$$RTG$$

```
> sfgmain(RTG,VV,VR);
```

$$-\frac{R(-gm+S\,C)}{1+RSC}$$

Example 15: [6]

Refer to Fig(2.9)

```
> gencsfg(begininput,Vi,1,2,sp,R4,4,3,sp,R2,2,3,sp,R1,5,2,sp,
  R3,4,5,sp,G,5,4,gm, Vg,sp,Vg,1,4,endinput);
```

$$RTG$$

```
>  sfgmain(RTG, VVi, VR1+VR2);
```

$$-\frac{R2\,R3 - R1\,R4}{R4\,R3 + R2\,R3 + R1\,R4 + R2\,R1}$$

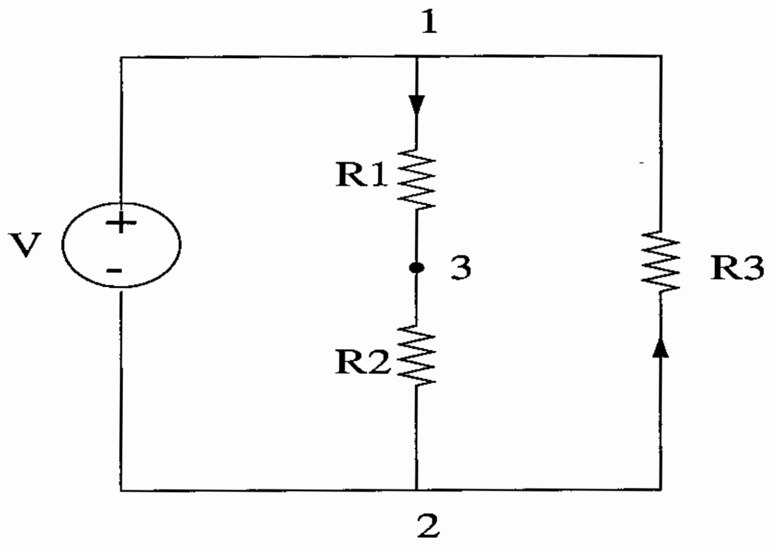


Figure 5.23: Circuit For Example 12

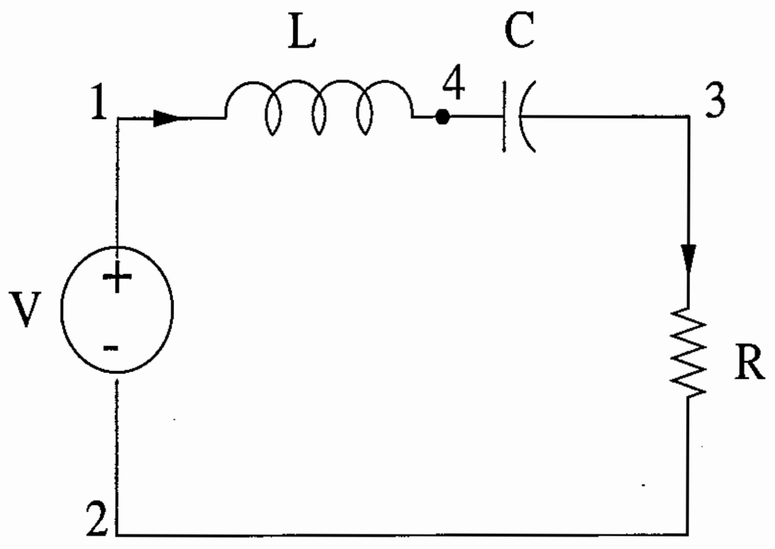


Figure 5.24: Circuit For Example 13

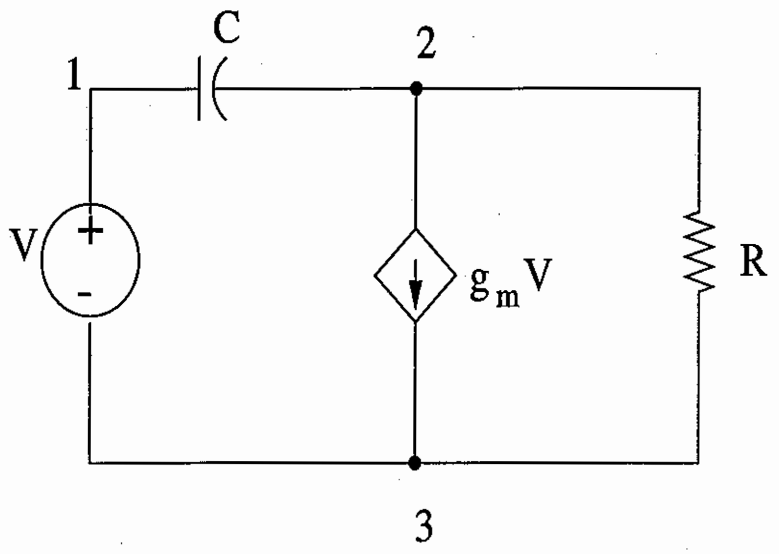


Figure 5.25: Circuit For Example 14

5.2 the mna Package

The *mna* package provides one function, *mnamain*. Here is the syntax of *mnamain*.

mnamain

- Calling Sequence

  mnamain begininput,element, +node, -node, sp, element, +node, -node, sp, . . . , endinput

- Parameters

  element is the circuit element, +node is a number indicating the positive end of the corresponding element, -node is a number indicating the negative end of the corresponding element (note that you should give highest node number to the ground node), and the placeholder, sp is used to separate the elements in the circuit. The element syntax is :

  R, L, C: R(L, C)(string), +node, -node

  independent voltage source: V(string), +node, -node

  independent current source: J(string), +node, -node

  voltage controlled voltage source: E(string), +node, -node, +controlled node, -controlled node, controlling parameter, element whose voltage is controlling voltage

  current controlled current source: F(string), +node, -node, +controlled node, -controlled node, controlling parameter, element whose current is controlling current

  voltage controlled current source: G(string), +node, -node, +controlled node, -controlled node, controlling parameter, element whose voltage is controlling voltage

  current controlled voltage source: H(string), +node, -node, +controlled node, -controlled node, controlling parameter, element whose element is controlling current

  operation amplifier: O(string), +inputnode, -inputnode, +outputnode, -outputnode

- Description

  mnamain computes each nodal voltages of the circuit. You can find the transfer function using these voltages.

The following maple worksheet presents some examples of *mnamain*.

Example 16:

Refer to Fig(5.1), But switch the nodes 2 and 3.

```
> mnamain('begininput', V,1,3,sp,R1,1,2,sp,R2,2,3,'endinput');
```

$$[V1 = V,\ V2 = \frac{V\,R2}{R1 + R2},\ iV = -\frac{V}{R1 + R2}]$$

Example 17:

Refer to Fig(5.25)

```
> mnamain('begininput',V,1,3,sp,C,1,2,sp,R,3,2,sp,G,2,3,1,3,gm,V,
  'endinput');
```

$$[V1 = V,\ V2 = -\frac{(gm - s\,C)\,V\,R}{1 + R\,s\,C},\ iV = -\frac{V\,(gm\,R + 1)\,s\,C}{1 + R\,s\,C}]$$

Example 18:

Refer to Fig(5.26)

```
> mnamain('begininput',R5,1,2,sp,R6,2,3,sp,R7,3,4,sp,R8,4,5,
  sp,R9,5,6,sp,J10,6,1,sp,O1,1,3,4,6,sp,O2,5,3,2,6,'endinput');
```

$$[V1 = J10\,(R5 + R6 + R7 + R8 + R9),\ V2 = J10\,(R6 + R7 + R8 + R9),$$
$$V3 = J10\,(R7 + R8 + R9),\ V4 = J10\,(R8 + R9),\ V5 = J10\,R9]$$

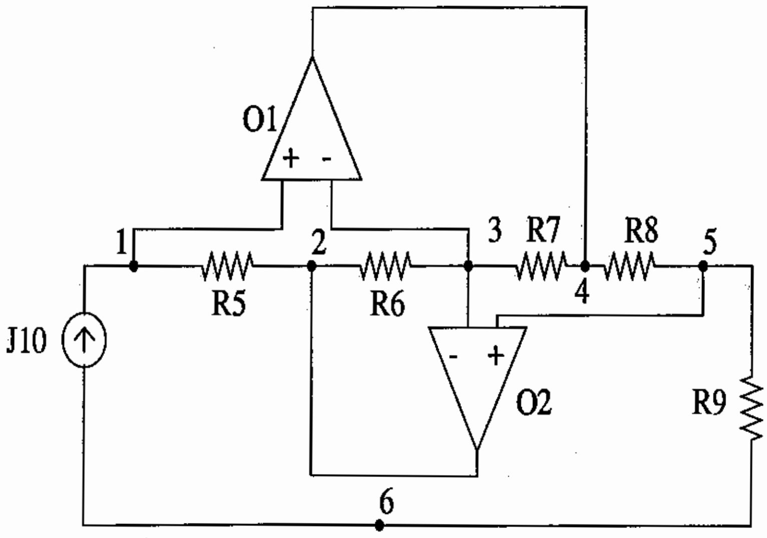


Figure 5.26: Circuit For Example 18

Chapter 6

Conclusions And Future Work

6.1 Conclusions

This thesis developed two packages *sfg* and *mna* for symbolic circuit analysis. Comparing these two packages, we can find the following conclusions:

- In sfg, we developed efficient techniques for finding all paths and all loops in the circuit. We use *the compact signal flowgraph* instead of *the primitive signal flowgraph*. The former is more powerful. We can also handle multi-input and multi-output transfer functions.

- In mna, we built models for circuit elements including opamp. Using maple, we get the nodal voltages directly.

- The operations needed in the two packages increases exponentially with the number of circuit nodes n. This is caused by the algorithms of the packages.

- The expression of transfer functions given by the two packages is not cancellation-free. That is to say, the expression has terms which can be cancelled. So the result is tedious sometimes. And the number of symbolic terms increase exponentially with the number of circuit node n. The *not cancellation-free* and *exponential growth* contribute to the circuit size limitation.

- *mna* is faster than *sfg* when handling small circuits as we see by directly comparing the executing times in maple.

- mna can handle circuits with nodes about 10. The sfg can handle circuit with nodes no more than 20.

limitation of sfg: No program can handle all types of problems. There are some circuits *sfg* can't handle. We use the following examples to explain this.

Example1:

Refer to Fig(6.1)

```
> gencsfg(begininput, J12,1,2,sp,Rb,2,4,sp,Rc,4,3,sp,Ra,3,2,sp,
  V31,3,1,endinput);
```

$$RTG$$

```
> sfgmain(RTG,VV31, VRa);
```

$$0$$

You get zero because VV31 is not in the compact sfg , VV31 is not involved in any loop. That means the coefficient of Z will be ZERO. You will get ZERO result. This is the limitation of the method. But you can calculate transfer function for iJ12. Once you get the following result, you can use Ohm's Law to get the transfer function for voltage scource.

```
> sfgmain(RTG,iJ12, iRa);
```

$$-\frac{Rc+Rb}{Ra+Rc+Rb}$$

Example2:

Refer to Fig(6.2)

```
> gencsfg(begininput, G12,1,2,gm,V31,sp,Rb,2,4,sp,Rc,4,3,sp,Ra,3,2,
  sp,V31,3,1,endinput);
```

Error, (in addedge) not a vertex –VV31"

Still because VV31 is not in the compact sfg, so you got the above error message.

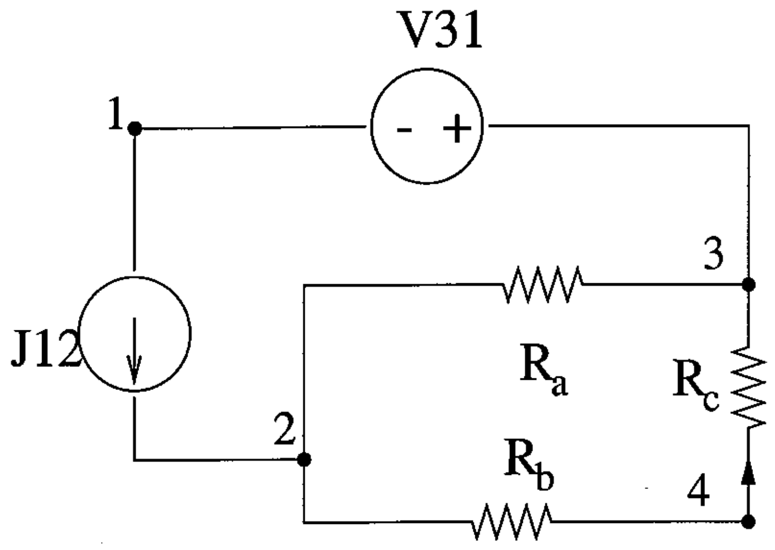


Figure 6.1: Circuit For Example 1

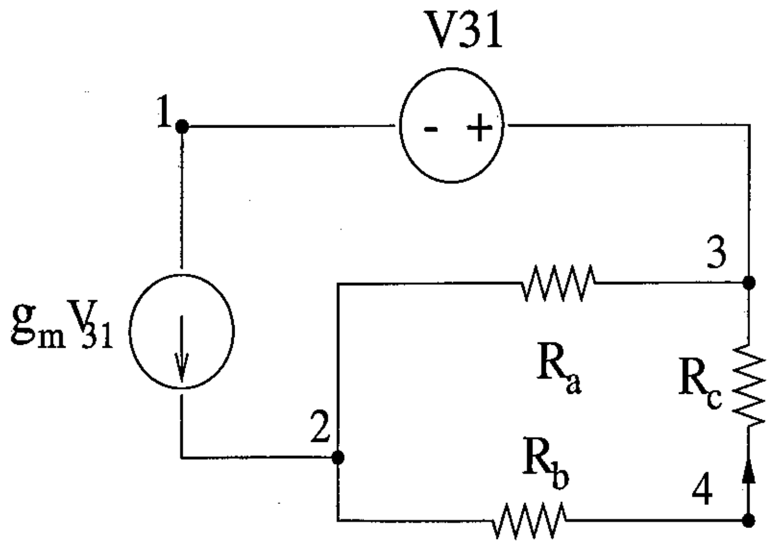


Figure 6.2: Circuit For Example 2

6.2 Future Work

First of all, we need to improve the method to make it *cancellation-free*. Then based on the packages, we can realize partitioning the circuit [26]. In this way, the packages will can handle large-scale circuit.

Appendix A

Source Code for sfg

```
sfg:=module()
     export sfgmain, gencsfg, ccgraph,ctree,fpath,f1loop,f2loop,fnloop,
            parsinp;
     local incmat, extractvw,dump,ind2l, rop, dig2udig, rsort,
           miniohm, flrr, extractvi, mykvl,kcl, kvl, compsfg,
          ohm, codingsfg, w1loop, wbranch, cdc, mycdc,weight2b;

#ccgraph generates a circuit graph for the input elements. It calls
#"dump" to dump out the input elements and then create the circuit
#graph.The weights of the edges of the graph are the input elements.

ccgraph:=proc()
     local L,N,vset,eset,i;
     L:=dump(args);
     N:=L[1];
     vset:={};
     eset:={};
     for i from 1 to N do
          vset:=vset union convert(L[2][i][2],set);
          vset:=vset union convert(L[2][i][3],set);
          eset:=eset union {[L[2][i][2],L[2][i][3]]};
     end do;
```

```
          new(G);
          addvertex(vset,G);
          for i from 1 to N do
               addedge([ [L[2][i][2],L[2][i][3]] ],weights=[L[2][i][1]],G);
          end do;

          return (G,eset);
     end proc;

     #cdc calls another function named "mycdc" to calculate the
     #deltac by the method in [6].

     cdc:=proc(IG::graph)
          local tpcdc,highest,sumloop,N,tpdc,i,tpitem,deltac;
          tpcdc:=mycdc(IG);
          highest:=tpcdc[1];
          sumloop:=tpcdc[2];
          #get the number of the highest order loop in the graph. And also
          #get the sum of the n-th order loop weights, where n is the index
          #of sumloop.

          if evalb(highest=0)=true then
               return (1);
          end if;
          #there is no loop in the graph at all.

          N:=highest;tpdc:=0;
          for i from 1 to N do
               tpitem:=(-1)^i*sumloop[i];
          tpdc:=tpdc+tpitem;
     end do;

     deltac:=1+tpdc;

     return(deltac);
end proc;

#codingsfg reads in a SFG whose vertices are the voltages and the
#currents of the original circuit diagrams. It outputs a graph whose
#vertices are coded into 1..N. It also outputs the coding scheme used.
#the scheme is simple. It uses the index of the returned list as the
#coded vertices of the corresponding vertices in the inputed SFG.

#it also add the -Z weight from the output vertex to the input vertex
#to form a closed SFG.

codingsfg:=proc(SFG::graph,INI,OUT)
     local vset,NV,vlist,codvset,elist,NE,i,fvb,svb,fv,sv,
          te,tw,iniv,outv;

     vset:=vertices(SFG);
     #get all the vertices.
     NV:=nops(vset);
     vlist:=convert(vset,list);
     #make the vertices to a list for coding's purpose.
     iniv:=rop(vlist,INI);
     outv:=rop(vlist,OUT);
     #get the corresponding coded number for the input vertex
     #and the output vertex for forming the closed SFG's purpose.
```

```
        codvset:={seq(i,i=1..NV)};
        #coded vertices.

        new(RG);
        addvertex(codvset,RG);

        elist:=convert(ends(SFG),list);
        #original edges list.
        NE:=nops(elist);
        for i from 1 to NE do
             fvb:=elist[i][1];
             svb:=elist[i][2];
             #the two ends of the edge i in elist.
             fv:=rop(vlist,fvb);
             sv:=rop(vlist,svb);
             #get the coded number for these two ends.
             te:=edges([fvb,svb],SFG);
             tw:=eweight(op(te),SFG);
             #get the weight for this edge in original graph.
             addedge([fv,sv],weights=tw,RG);
        end do;

        addedge([[outv,iniv]],weights=-Z,RG);
        #form the closed SFG

        return(RG,vlist)
   end proc;


   #compsfg implements the compact generating method in [6].
```

```
#the input to this function is a circuit diagram generated by "ccgraph".
#the output is the compact SFG of this circuit.
#the controlled elements in the CSFG are represented as their indepen-
#dent equivalences. And also, the resulting CSFG is not the closed SFG
#and also does not include the relationship between the controlling
#elements and the controlled elements. This is the task fulfilled in the
#"gencsfg" function.
#it calls a program "ctree", which is used to find tree and cotree.
#and also it calls several other programs "ohm", "kcl", "kvl", "extractvi",
#"flrr". See the comments of these programs for their function.

compsfg:=proc(G::graph)
     local OH,TV,CTC,KVL,KCL,NTV,j,eq,lhev,res,isub,ires,req,fres,NI,
          addvset,addwset,k,NCTC,NV,resctree,Tree,CoTree;

     resctree:=ctree(G);
     Tree:=resctree[1];
     CoTree:=resctree[2];
     #generate the tree and the cotree. There are surely V,E,H in the
     #tree branches and J,F,G branches in the cotree branches. And also
     #the tree should be connected at this stage. These can be seen
     #in the implementation of function "ctree".

     draw(Tree);
     draw(CoTree);

     OH:=ohm(G,Tree,CoTree);
     TV:=OH[1];
     CTC:=OH[2];
     #get the Ohm's Law for the weights.
```

```
#for the tree branches, the Ohm's Law is expressed as Vxx=xx*Ixx,
#while for the cotree branches, the Ohm's Law is expressed as
#Ixx=xx*Vxx. NOTE: the sources VEH and JFG are not in the equations
#returned by the Ohm's Law.

KVL:=kvl(Tree,CoTree);
#generate the KVL equations for the cotree voltages. Due to the
#fact that there is no way to guarantee that the loop formed to
#generate the KVL equation will surely include the voltage sources
#VEH, as the result, there is no way to guarantee the equations
#returned by "kvl" will surely have the voltage sources VEH at the
#rhs of them. The result is that if they are not in the rhs of the
#equations, they will not appear in the CSFG formed later. There
#should be some mechanism to check whether they are in the CSFG
#at last. If they are, then it is ok. If they are not, then they
#should be added to the CSFG if the user input wants to calculate
#them. They should not be added to the CSFG if the user does not
#want to calculate them because the user does not want to calculate
#them. And if they are added to the CSFG, then the closed CSFG will
#also not add an edge to them. Therefore, the closed CSFG which
#is needed to calculate the Deltac will not be connected any
#more.
#Therefore, as the result, the work of adding them should be done
#at the parse input and then coding stage, not before it.

KCL:=kcl(G,Tree);
#get the kcl equations for the tree voltages.
#similar with the "kvl" function, there is no guarantee that
#the current sources will appear at the rhs of the kcl equations.
#If they are not, then they will not appear in the CSFG formed later.
```

```
#They should also be judged whether in the CSFG at the stage of
#parsing the input and forming the closed CSFG.

new(RTG);

NTV:=nops(TV);
for j from 1 to NTV do
     eq:=TV[j];
     lhev:=lhs(eq);
     res:=extractvi(eq);
     isub:=op(res[1]);
     ires:=flrr(KCL,isub);
     req:=subs(isub=ires,eq);
     fres:=extractvi(req);

     NI:=nops(fres[1]);
     addvset:=fres[1];
     addwset:=fres[2];
     for k from 1 to NI do
          addvertex({lhev,addvset[k]},RTG);
          addedge([[addvset[k],lhev]],weights=addwset[k],RTG);
     end do;
end do;
#generate part of the CSFG according to the first set of Ohm's Law
#which are in the form as Vxx=xx*Ixx. Because in Ohm' Law, only
#RLC have Ohm's Law equations, their Ixx should be in the kcl
#equations from function "kcl".

NCTC:=nops(CTC);
for j from 1 to NCTC do
```

```
        eq:=CTC[j];
        lhev:=lhs(eq);
        res:=extractvi(eq);
        isub:=op(res[1]);
        ires:=flrr(KVL,isub);
        req:=subs(isub=ires,eq);
        fres:=extractvi(req);

        NV:=nops(fres[1]);
        addvset:=fres[1];
        addwset:=fres[2];
        for k from 1 to NV do
            addvertex({lhev,addvset[k]},RTG);
            addedge([[addvset[k],lhev]],weights=addwset[k],RTG);
        end do;
    end do;
    #generate the other part of the CSFG from the other set of ohm's
    #law equations. These equations are in the form as Ixx=xx*Vxx.
    #The Vxx of these RLC should in the kvl equations from the
    #function "kvl".

    #ATTENTION: Because the compact SFG generated in this way comes
    #from the Ohm's Law equations. But there will surely be no VEH and
    #JFG in these equations. And also, in the equations from "kvl" and
    #"kcl", there is no way to guarantee that the VEH and JFG will
    #appear in the equations returned by these two functions, therefore,
    #there is no guarantee that VEH and JFG will appear in the compact
    #SFG generated in this function. There should be some way to check
    #this kind of situation. But the checking should only be done at
    #the stage of parsing the input and then generating the closed CSFG.
```

```
     #If the user does not input the VEH or JFG, and they are added to
     #the CSFG, then when forming the closed CSFG, there will be no edges
     #incident with these sources. Therefore, the graph transferred to
     #calculate the determinant Deltac will not be connected any more.
     #Therefore, if the user input these sources but they are not in
     #the CSFG, then they should be added.

     return(RTG);
end proc;

#ctree is used to find tree and cotree using P.M.Lin method.
#Input of ctree here is a graph, output are tree and cotree.

#ATTENTION: this function can guarantee that the voltage sources (VEH)
#are in the tree branches and the current sources (JFG) are in the
#cotree branches.

ctree:=proc(G::graph)
     local N,en,ew,ind,ent,ver,counter1,counter2,k,enttype,tb,tbe,cb,
          cbe,temp1,temp2,l,vertT,vertCT,i,UT,fv,sv,te,res,tw;
     N:=nops(edges(G));
     en:=ends(G);
     ew:=eweight(G);
     ind:=indices(ew);
     ent:=entries(ew);
     ver:=vertices(G);

#The following for loop is used for finding Voltage sources and Current
#Sources out, and recording corresponding branches.
```

```
counter1:=0;
counter2:=0;
for k from 1 to N do
     enttype:=substring(op(ent[k]) ,1);
     if enttype='V' or enttype='E' or enttype='H' then
          counter1:=counter1+1;
          tb[counter1]:=op(ends(ind[k],G));
          tbe[counter1]:=op(ent[k]);
     elif enttype='J' or enttype='F' or enttype='G' then
          counter2:=counter2+1;
          cb[counter2]:=op(ends(ind[k],G));
          cbe[counter2]:=op(ent[k]);
     end if;
end do;

temp1:={seq(tb[i],i=1..counter1)};
temp2:={seq(cb[i],i=1..counter2)};
l:=en minus temp1 minus temp2;
#the above lines are used to find the left branches of the
#graph after extracting the voltage and current sources.

vertT:=map(op,temp1);
vertCT:=map(op,temp2);

new(T);
addvertex(vertT,T);
for i from 1 to counter1 do
     addedge(tb[i],weights=[tbe[i]],T);
end do;
#construct the primitive tree.
```

```
    new(CT);
    addvertex(vertCT,CT);
    for i from 1 to counter2 do
         addedge(cb[i],weights=[cbe[i]],CT);
    end do;
    #construct the primitive cotree.
    for i from 1 to N-counter1-counter2 do
         UT:=dig2udig(T);
         fv:=l[i][1];
         sv:=l[i][2];
         tb:=[l[i][1],l[i][2]];
         te:=op(edges(l[i],G));

         res:=fpath(UT,fv,sv);
         tw:=eweight(op(edges(l[i],G)),G);
         if res[1]=0 then
              addvertex({fv,sv},T);
              addedge(tb,weights=tw,T);
         else
              addvertex({fv,sv},CT);
              addedge(tb,weights=tw,CT);
         end if;
         #judge the branches of RLC to add them either
         #to the tree or cotree.
    end do;

    return (T,CT);
end proc;
```

```
#dig2udig reads in a directed graph and then output an undirected
#graph which has the same topological structure as the input graph.
#it does not preserve the weights of the original graph because some
#times it is impossible to do so.

dig2udig:=proc(G::graph)
    local vsetorig,esetorig,bset,N,i,tb;
    vsetorig:=vertices(G);
    esetorig:=edges(G);
    bset:={};
    N:=nops(esetorig);

    for i from 1 to N do
        tb:={convert(ends(esetorig[i],G),set)};
        bset:=bset union tb;
    end do;

    new(UG);
    addvertex(vsetorig,UG);
    addedge(bset,UG);
#attention here, {1,2} and {2,1} are the same to addedges and also
#addedges will not repeatedly add {x,y}
    return (UG);
end proc;

#dump dumps out the inputed values for the later usage.
#it calls another program "ind2l",which covert a table to ordered list.
#user input syntax: 'beginput', elementname, (contrpos,contrneg),pos,
#neg,'sp',elementname,(contrpos,contrneg),pos,neg,'sp','endinput'.
```

```
dump:=proc()
     local gnum,enum,i,inp,totale;
     if args[1]<> begininput or args[nargs]<> endinput then
          eror("syntax error");
     end if;

     gnum:=1;enum:=1;
     for i from 2 to nargs-1 do
          if args[i] <> sp then
               inp[gnum][enum]:=args[i];
               enum:=enum+1;
          else
               gnum:=gnum+1;
               enum:=1;
          end if;
     end do;

     totale:=gnum;

     return(totale,[seq(ind21(op(eval(inp[i]))),i=1..gnum)]);

end proc;

#extractvi parses an equation as ia=C1*Va+C2*Vb... It is used to extract
#the V variables and the coefficients of these variables. It can also be
#used to extract the i variable in the V=C1*ia+C2*ib....

extractvi:=proc(EQUA::equation)
     local EQ,lhseq,lhstype,rhseq,rhslist,counter,NR,j,ope,NOPE,k,
           opelement,optype,varr,coff,varlist,coflist;
```

```
      EQ:=expand(EQUA);
      lhseq:=lhs(EQ);
      lhstype:=substring(lhseq,1);
      rhseq:=rhs(EQ);
      rhslist:=[op(rhseq)];
      counter:=0;

      NR:=nops(rhslist);
      for j from 1 to NR do
           ope:=[op(rhslist[j])];
           NOPE:=nops(ope);
           for k from 1 to NOPE do
                opelement:=ope[k];
                if type(opelement,numeric)=true or
                     type(opelement,`^`)=true then
#numeric type is for the numeric coefficients and `^` type is for the
#reciprocal coefficients. Note the back quotes used for `^`.
                     optype:='Z';
                else
                     optype:=substring(opelement,1);
                end if;
                if lhstype='i' then
                     if optype='V' then
                          if has(eval(varr),opelement)=false then
                               counter:=counter+1;
                               varr[counter]:=opelement;
                               coff[counter]:=coeff(rhseq,
                                          opelement);
                          end if;
                     end if;
                elif lhstype='V' then
                    if optype='i' then
                        if has(eval(varr),opelement)=false then
                            counter:=counter+1;
                            varr[counter]:=opelement;
                            coff[counter]:=coeff(rhseq
                                        ,opelement);
                        end if;
                    end if;
                end if;
            end do;
        end do;

        varlist:=ind21(op(eval(varr)));
        coflist:=ind21(op(eval(coff)));
        return(varlist,coflist);
    end proc;

#extractvw is a general implementation of another function called
#"extractvi". This function extracts all the operands of an equation
#if the operands are V(voltage) or i (current). It also return the
#coefficients.

#it reads in an equation and outputs three things. The first is a
#symbol which is the left hand side (lhs) of the equation. The second
#is the V and i items in the rhs of the equation. The third are the
#corresponding coefficients for the operands.

extractvw:=proc(EQUA::equation)
    local EQ,lhseq,rhseq,rhslist,counter,NR,j,ope,NOPE,k,opelement,
     optype,varr,coff,varlist,coflist;
EQ:=expand(EQUA);
#the equation needs to be expanded in order to find the coefficient
#of an operand correctly.

lhseq:=lhs(EQ);
rhseq:=rhs(EQ);
rhslist:=[op(rhseq)];
#rhlist is the list for the rhs eq's operands. Note that there are
#composite operands in rhslist.

counter:=0;
NR:=nops(rhslist);
for j from 1 to NR do
     ope:=[op(rhslist[j])];
     #Because there are composite operands in rhlist, it needs
     #to judge each single operand one by one.

     NOPE:=nops(ope);
     for k from 1 to NOPE do
          opelement:=ope[k];
          #the single operand to be judged.

          if type(opelement,numeric)=true or
               type(opelement,`^`)=true then
               optype:='Z';
          #surely if the operand is a number or a
          #reciprocal, there is no substring. 'Z'
          #is just for convenience. It can be any
          #meaningless chars.
```

```
                else
                     optype:=substring(opelement,1);
                end if;

                if optype='i' or optype='V' then
                     if has(eval(varr),opelement)=false then
                          counter:=counter+1;
                          varr[counter]:=opelement;
                          coff[counter]:=coeff(rhseq,opelement);
                          #records the vars and the
                          #corresponding coefficients.
                     end if;
                end if;
           end do;
      end do;

      varlist:=ind2l(op(eval(varr)));
      coflist:=ind2l(op(eval(coff)));
      return(lhseq,varlist,coflist);
end proc;

#f1loop finds out the 1st order loops of the given graph.
#When there are no 1st order loops in the given graph, then 0 and an
#empty list is returned.

f1loop:=proc(G::graph)
     local TG,N,counter,k,L,j,first_loop,res,i;
     TG:=duplicate(G);
     N:=nops(vertices(TG));
```

```
    counter:=0;
    for k from 1 to N do
         L:=fpath(TG,k,k);
         #path from k to k is a loop.
         if L[1]<>0 then
              for j from 1 to L[1] do
                   counter:=counter+1;
                   first_loop[counter]:=L[2][j];
              end do;
         end if;
         delete(incident(k,TG),TG):
         #deleting the branched connected with k, in order to avoid
         #duplication.
    end do;
    #res:=[seq(0,i=1..counter)];
    res:=array(1..counter);
    #here I use array not seq because res is a long list sometimes.
    #while seq has size limitation.
    for i from 1 to counter do
         res[i]:=first_loop[i];
    end do;

    return(counter,eval(res));

end proc;

#f2loop finds out the 2nd order loops of the given graph. It finds
#out the 2nd order loop by comparing the table of the 1st order loops.

f2loop:=proc(IG::graph)
```

```
local G,L,N,counter,cjlist,i,seta,counterj,j,setb,res,sec_loop;
G:=duplicate(IG);
L:=f1loop(G);
N:=L[1];counter:=0;

if N=0 then
     return(0,[],[]);
end if;

cjlist:=array(1..N,[seq(0,i=1..N)]);
for i from 1 to N-1 do
     seta:=convert(L[2][i],set);
     counterj:=0;
     for j from i+1 to N do
          setb:=convert(L[2][j],set);
          res:=seta intersect setb;
          #compare the i-th 1st loop with its higher
          #order loops to see whether they have
          #intersected nodes.

          if nops(res)=0 then
               counterj:=counterj+1;
               counter:=counter+1;
               sec_loop[i][counterj]:=j;
          end if;
     end do;
     if counterj=0 then
          sec_loop[i][1]:=NULL;
     end if;
     cjlist[i]:=counterj;
```

```
          end do;
          sec_loop[N][1]:=NULL;

return (counter,eval(cjlist),[seq(ind2l(op(eval(sec_loop[i]))),
               i=1..N)]);
end proc;

#the input to flrr is a set of equations and the lhs of an
#equation which is to be found in the set. The output of the function
#is the rhs of the found equation, if there is any. If there is no such
#equation in the set, then FAIL is returned.

flrr:=proc(ES::set,l2f)
     local N,i;
     N:=nops(ES);
     for i from 1 to N do
          if evalb(lhs(ES[i])=l2f)=true then
               return(rhs(ES[i]));
          end if;
     end do;
     return(FAIL);
end proc;

#fnloop is used to find n-th order loops of the given graph.
#it calls f2loop and then f2loop calls f1loop.

#this function uses the similar algorithm as finding the path.
#When see the figure 14-11 on page 555 of [6], this figure can be
#rotated counter clock wise 90 degree. Then this figure is similar
#with the figure 14-9 on page 552 of [6]. The difference is the
```

```
#terminating condition is not the same.

fnloop:=proc(G::graph)
     local L,N,counterj,T,i,current,parent,flag,w,vnode,j,P,
              cj_list,cj,child,TL,tempp,bool,retres,kk,k,
              counter;


     L:=f2loop(G);
     if L[1]=0 then
          return(0,[],[]);
     end if;
     #L[1]=0 means there is no second order path. So no n-th order loop.
     N:=nops(convert(L[2],list));
     #because the type of L[2] is not list before L[2] is converted.
     counterj:=array(1..N,[seq(0,k=1..N)]);
     counter:=0;
     #counter is the number of how many tree branch.
     #counterj is a list which will record the number of tree branch
     #of each loop.
     T[1]:=[op(L[3][1]),-1];
     for i from 2 to N do
          T[i]:=[op(L[3][i]),0];
     end do;
     #similar to routing table in fpath.
     #-1 is used to indicate the current row.
     #0 is used to indicate the end of one row.

     #begin to find loop.
     for i from 1 to N do
          current:=i;parent:=i;flag:=0;
```

```
#we use parent, current and child to locate the node.
#flag is used to check if there is high order loop.
if i <>1 then
     T[i]:=subsop(nops(T[i])=-1,T[i]);
     T[i-1]:=subsop(nops(T[i-1])=0,T[i-1]);
end if;
#to reset the flag of current row.
w:=1;vnode:=2;j:=i;P[i][1][1]:=i;
cj_list:=array(1..N,[seq(1,k=1..N)]);
#ci_list is used to record the colum which has been traversed.

while true do
     cj:=cj_list[j];
     P[i][w][vnode]:=T[j][cj];
     child:=P[i][w][vnode];

     if child >0 then
          TL:=op(eval(P[i][w]));
          tempp:=ind21(TL);
          bool:=evalb(numboccur(tempp,P[i][w][vnode])>1);
          #bool is used to check if the node is repeated.
          if bool=true then
               j:=P[i][w][vnode-1];
               cj_list[j]:=cj_list[j]+1;
          else
          #if node is repeated, back one node.
          #and go to next column.
               if current=i then
                    parent:=current;
                    current:=child;
                    j:=P[i][w][vnode];
                    vnode:=vnode+1;
               #current=i mean the current node is the root
               #node.change current to next node.
               else
                    if has(child,L[3][parent])=true then
                         parent:=current;
                         current:=child;
                         j:=P[i][w][vnode];
                         vnode:=vnode+1;
                    #to judge if the child found has the same
                    #parent with the current node.
                    else
                         j:=P[i][w][vnode-1];
                         cj_list[j]:=cj_list[j]+1;
                    #if child has no same parent with the
                    #current node, then back one node and
                    #go to next column.
                    end if;
               end if;
          end if;
     elif child =0 then
          j:=P[i][w][vnode-2];
          cj_list[j]:=cj_list[j]+1;
          cj_list[P[i][w][vnode-1]]:=1;
          current:=parent;
          #if reach the end of one row, then back two nodes,
          #and parent becomes current.
          if (vnode-3)<=0 then
               parent:=P[i][w][1];
          else
               parent:=P[i][w][vnode-3];
          end if;
          #find new parent.
          if nops(T[P[i][w][vnode-1]])=1 then
                    flag:=1;
                    counterj[i]:=counterj[i]+1;
                    counter:=counter+1;
                    retres[i][counterj[i]]:=
                                   ind21(op(eval(P[i][w])));
                    for kk from 1 to vnode-2 do
                         P[i][w+1][kk]:=P[i][w][kk];
                    end do;
                    w:=w+1;
                    vnode:=vnode-1;
          #nops(T[P[i][w][vnode-1]])=1) is the condition of
          #terminating,i.e. we reach tips. Then we need give
          #the previous vnode-2 nodes to next tree branch.
          else
                    P[i][w][vnode]:=NULL;
                    vnode:=vnode-1;
          #if not reach tips, then look for vnode-1.
          end if;
     elif child =-1 then
          if flag=0 then
               retres[i][1]:=0;
          end if;
          break;
     end if;
end do;
```

```
     end do;
     return (counter,eval(counterj),[seq(ind21(op(eval(retres[i]) ) ),
             i=1..N)]);
end proc;

#fpath is one of the core functions in the package. It finds the
#path from the inputed initial vertex to the inputed last vertex. When
#these two vertices are the same, then it finds out all of the loops of
#the given vertex.

#it calls "ind21" and "rop", see the comments of them to get their
#function.

#tested examples 1. the model in the book;
#                2. petersen() in the maple;
#                3. tetrahedron() in the maple;
#                4. octahedron() in the maple but failed;
#                   due to the reason that it has a "0" vertice;

fpath:=proc(G::graph,INI::integer,LAS::integer)
     local dep,op_table,k,i,v,N,w,vnode,j,P,cj,counter,tempp,tempp2,
          bool,TL,TL2,cj_list,vset;
     dep:=departures(G);
     vset:=vertices(G);
     op_table:=op(op(dep));
     k:=max(op(vset));

     if has(vset,INI)=false or has(vset,LAS)=false then
          return(0,[]);
     end if;
```

```
          for i from 1 to k do
               if has(vset,i)=true then
                    v[i]:=convert(dep[i],list);
                    v[i]:=sort(v[i],'>');
                    if i=INI then
                         v[i]:=[op(v[i]),-1];
                    else
                         v[i]:=[op(v[i]),0];
                    end if;
               else
                    v[i]:=[0];
               end if;
          end do;
     #The end of generating the routine table.

     #The beginning of the path finding algorithm.
          N:=max(op(vset));
          w:=1;vnode:=2;j:=INI;P[1][1]:=INI;
          counter:=0;cj_list:=[seq(1,i=1..N)];
     #counter is used  to record the number of path.
     #cj_list is a flag used to record the column which has
     #been traversed in routing table. Its value  is transfered
     #to cj (column) in order to take out the path node.
          while true do
          #PF2
               cj:=cj_list[j];
               P[w][vnode]:=v[j][cj];
               #PF3
               if P[w][vnode] >0 then
```

```
#PF4
TL:=op(eval(P[w]));
tempp:=ind2l(TL);
bool:=evalb(numboccur(tempp,P[w][vnode])>1);
# bool is used to judge if the node is traversed
# more than one time.
     if P[w][vnode] = LAS then
          counter:=counter+1;
          j:=P[w][vnode-1];
          # back one node if find a path successfully.
          cj_list[j]:=cj_list[j]+1;
          #column flag increases one
          for k from 1 to vnode-1 do
               P[w+1][k]:=P[w][k];
          end do;
          #give the first vnode-1 node to next path.
          w:=w+1;
          #increase path number.
     elif bool=true then
          j:=P[w][vnode-1];
          #bake one node if the vnode-th node is repeated.
          cj_list[j]:=cj_list[j]+1;
          #column flag increases one
     else
          j:=P[w][vnode];
          vnode:=vnode+1;
          #go ahead to find next node.
     end if;
     #call PF2
#PF5
```

```
            elif P[w][vnode] =0 then
                #P[w][vnode]=0 means that one row of routing table has
                #been traversed.
                j:=P[w][vnode-2];
                #back two nodes.
                cj_list[j]:=cj_list[j]+1;
                #column flag increases one.
                cj_list[P[w][vnode-1]]:=1;
                #the above line is used to reset the column flag
                #in order to traversed again.
                P[w][vnode]:=NULL;
                TL2:=op(eval(P[w]));
                tempp2:=ind21(TL2);
                #ind21 can remove the NULL element.
                for k from 1 to vnode-1 do
                    P[w][k]:=tempp2[k];
                end do;
                #the the first k nodes of w-th path.
                vnode:=vnode-1;
                #call PF2
            elif P[w][vnode] <0 then
                #P[w][vnodde] <0 in the program I give -1 to it,
                #means that you finish the whole process of finding
                #path.
                break
            end if;
        end do;

        return (counter,[seq(ind21(op(eval(P[k]))),k=1..counter)]);
    end proc;
```

```
#gencsfg implements the first step in the SFG package. The inputs to
#this function can be only R, L, C, E (VCVS), H (CCVS), V, F (CCCS),
#G(VCCS) and J (independent current source).

#the input of the function is same as dump input, i.e.'begininput',...
#'endinput'.
#the output of the function is CSFG.

#For the independent sources, this function itself makes no changes on
#the compact sfg (CSFG). It just outputs the CSFG.
#For controlled sources, it first replace the controlled sources by their
#equivalent independent sources and then generate the circuit with the
#replaced vertices. Before it generates the circuit diagram, this
#function also records the expression of the relationship between the
#independent source and the controlled source. The controlled source can
#be controlled by more than one independent or RLC variables. Then the
#controlling variable is searched in the CSFG. If it or they are not in
#the CSFG, then Ohm's Law is used. Then add the new branches and the
#weights to the CSFG and then outputs the CSFG.

gencsfg:=proc()
    local L,circuitdig,compactsfg,inpres,NIN,inp,counter,expr,
        l,opelement,etype,newlhsv,lhsvlist,tprhs,NINPL,
        NLOOP,j,tpnewrhsv,tpnewrhsvlist,newrhsv,ctrleq,
        k,i,inpccgraph,vset,NC,ceqj,tplist,NR,tpv,feq,tpeq,
        extres,headnode,nodelist,weightlist,NN,flag;

    inpres:=dump(args);
    NIN:=inpres[1];
```

```
inp:=inpres[2];
#get the parsed inputs.

counter:=0;
#counter is used to record the number of controlled sources.

expr[1]:='begininput';
#expr is used to record the final input to generate the circuit
#diagram. The controlled sources in this input are replaced by
#their equivalents.

flag:=0;
#flag is used to indicate whether there is any controlled source
#in the circuit.

for l from 1 to NIN do
     opelement:=inp[l][1];
     etype:=substring(opelement,1);
     if etype='E' or etype='H' then
          flag:=1;
          counter:=counter+1;
          newlhsv:=cat('V',opelement);
          #the newly introduced vertex to represent
          #the controlled element.

          lhsvlist[counter]:=newlhsv;
     #If the inputed element is a E or H, then introduce a new
     #node which is composed by concatenating the V with the
     #element. This new node is used to generate the circuit
     #diagram.
```

```
#lhsvlist is for left hand side vertices list. It is
#used for storing all the newly introduced vertices.

    tprhs:=0;
    NINPL:=nops(inp[1]);
    NLOOP:=(NINPL-3)/2;
    #if the element is a E or H or F or G, then
    #the variables needed to describe these kinds of
    #elements are at least 5 variables according to
    #the stipulations. And also, the first three variables
    #are not useful to determine the algebraic expression
    #between the controlled elements with the controlling
    #elements.
    #For the variable beginning from the 4th elements for
    #each E, H, F, or G, each two node, i.e. a pair of
    #nodes represents the information of the controlled
    #elements. This is the reason that the number of
    #the loop, NLOOP, is (NINPL-3)/2.

    for j from 1 to NLOOP do
        if etype='E' then
            tpnewrhsv[j]:=cat('V',inp[1][2*j+3]);
        else
            tpnewrhsv[j]:=cat('i',inp[1][2*j+3]);
        end if;
        tprhs:=tprhs+inp[1][2*j-1+3]*tpnewrhsv[j];
    end do;
    #for each controlling elements, form the vertices
    #and the algebraic equation of the controlled element.
```

```
          tpnewrhsvlist:=ind2l(op(eval(tpnewrhsv)));
          newrhsv[counter]:=tpnewrhsvlist;
          #newrhsv is for new right hand side vertices. It
          #is used to store all the newly introduced vertices
          #for the controlling elements.

          ctrleq[counter]:=newlhsv=tprhs;
          #ctrleq is for controlling equations. It is used
          #to record each of the controlling equations between
          #the controlled element and the controlling elements.

          expr[2*l]:=opelement,inp[l][2],inp[l][3];
          expr[2*l+1]:='sp';
          #form the new input to "ccgraph", which generate
          #the circuit diagram according to the new replacements.

     elif      etype='F' or etype='G' then
     #this case is similar with the above case.

          flag:=1;
          counter:=counter+1;
          newlhsv:=cat('i',opelement);
          lhsvlist[counter]:=newlhsv;

          tprhs:=0;
          NINPL:=nops(inp[l]);
          NLOOP:=(NINPL-3)/2;
          for j from 1 to NLOOP do
               if etype='F' then
```

```
                        tpnewrhsv[j]:=cat('i',inp[l][2*j+3]);
                    else
                        tpnewrhsv[j]:=cat('V',inp[l][2*j+3]);
                    end if;
                    tprhs:=tprhs+inp[l][2*j-1+3]*tpnewrhsv[j];
                end do;
                tpnewrhsvlist:=ind2l(op(eval(tpnewrhsv)));
                newrhsv[counter]:=tpnewrhsvlist;

                ctrleq[counter]:=newlhsv=tprhs;


                expr[2*l]:=opelement,inp[l][2],inp[l][3];
                expr[2*l+1]:='sp';
            else
            #if the inputed elements are RLC, then simply form the
            #new input the same as the old inputs.

                expr[2*l]:=seq(inp[l][k],k=1..3);
                expr[2*l+1]:='sp';
            end if;
        end do;
        expr[2*NIN+1]:='endinput';

        inpccgraph:=seq(expr[i],i=1..2*NIN+1);
        #generate the new inputs to "ccgraph".

        L:=ccgraph(inpccgraph);
        #generate the circuit diagram from the inputs. The controlled
        #elements are represented as their independent equivalences.
```

```
circuitdig:=L[1];
#get the circuit diagram of the inputed elements.

compactsfg:=compsfg(circuitdig);
#Get the compact SFG from the circuit diagram.
#The compact sfg at this stage has only independent sources and
#RLC. The branches and the weights for the controlled elements need
#to be added in the following.

if evalb(flag=1)=true then
     vset:=vertices(compactsfg);
     NC:=nops([indices(lhsvlist)]);
#Get the number of all the newly introduced vertices for the
#controlled elements.

     for j from  1 to NC do
     #make judgments on each of these new vertices.

          ceqj:=ctrleq[j];
          #get the relationship between the controlled
          #vertices and the controlling vertices.

          tplist:=newrhsv[j];
          NR:=nops(tplist);
          #get the number of controlling elements for each
          #controlled elements.

          for l from 1 to NR do
     #judge whether these controlling elements are in the CSFG or
     #not.
```

```
                    tpv:=tplist[1];
                    if has(vset,tpv)=true then
                         feq:=ceqj;
                    else
                         tpeq:=miniohm(tpv);
                         feq:=subs(tpeq,ceqj);
                    end if;
               #if the controlling elements are not in the CSFG,
               #then Ohm's Law is used and then the relationship
               #is updated according to Ohm's Law.

               end do;
               extres:=extractvw(feq);
               headnode:=extres[1];
               nodelist:=extres[2];
               weightlist:=extres[3];
          #get the vertices (they are supposed to be all
          #in the CSFG at this stage), the branches and the weights.

               NN:=nops(nodelist);
               for l from 1 to NN do
                    addedge([nodelist[l],headnode],
                         weights=weightlist[l],compactsfg);
               end do;
          #add the branches and the weights of each of the
          #newly introduced vertices for the controlled elements.

          end do;
     end if;
     return (compactsfg);
end proc;

#incmat reads into a directed weighted graph and then output its
#incidence matrix together with a list which records the sequence of
#the column of the incidence matrix. The usage of the list is to avoid
#wrongly multiplication of the incidence matrix with the branch currents.

#If an edge is not incident to a vertex then the corresponding table
#entry is 0. If a directed edge is leaving a vertex then the corresponding
#table entry is 1. If a directed edge is coming a vertex then the correspo-
#nding table entry is -1.

incmat:=proc(G::graph)
     local vset,eset,NV,NE,inmatrix,collist,i,te,tailnode,headnode;
     vset:=vertices(G);
     eset:=edges(G);
     NV:=nops(vset);
     NE:=nops(eset);

     inmatrix:=Matrix(NV,NE);
     collist:=[seq(0,i=1..NE)];
     for i from 1 to NE do
          te:=eset[i];
          tailnode:=tail(te,G);
          headnode:=head(te,G);
          inmatrix[tailnode,i]:=1;
          inmatrix[headnode,i]:=-1;
          collist[i]:=[tailnode,headnode];
          #collist is a recorder to record the order of the columns
          #of the incidence matrix. It is used for later multiplication
          #with the currents.

     end do;
     return([inmatrix,collist]);

end proc;

#ind2l is also another core functions for the packages. It is important
#because the usage of table in other functions. In order to suitablely
#return those tables, this function can be used to convert the tables
#into ordered list. Further, for NULL entry in the table, this function
#will let it equal negative infinity. fpath will use this.

#this program calls another program "rop", which find the position of a
#given number in a list.

#the input of this function must be op(eval(T)), suppose that T is a one
#level table.

ind2l:=proc(LL::list)
     local counter,N,temp1,temp2,k,trhs,result,position,i;
     counter:=0;
     N:=nops(LL);
     temp1:=[seq(-1,i=1..N)];
     temp2:=[seq(-1,i=1..N)];
     for k from 1 to N do
          temp1[k]:=lhs(LL[k]);
          trhs:=rhs(LL[k]);
          if (trhs=NULL) then
               temp2[k]:=-infinity;
               counter:=counter+1;
          else
               temp2[k]:=trhs;
          end if;
     end do;

     result:=[seq(-1,i=1..N-counter)];
     for k from 1 to N-counter do
          position:=rop(temp1,k);
          result[k]:=temp2[position];
     end do;
     return (result);
end proc;

#kcl implements the KCL according to the method in [6]. The resulted
#equations are returned as a set of equations.

#the input to this function is a circuit diagram and the tree of the
#diagram.
#the output of this function is a set of kcl equations.

#similar with the "kvl" function, there is no guarantee that the current
#sources will appear at the rhs of the kcl equations. If they are not,
#then they will not appear in the CSFG formed later.

#it calls another program "incmat" which gives the incidence matrix
#of a graph. Before running the program, must type with(LinearAlgebra).
```

```
kcl:=proc(G::graph,T::graph)
     local incres,incmatrix,incmatcol,dirtbset,NTB,unknowni,counter,
          NE,ivector,k,te,tw,tpneww,neww,res,M,elist,i,eqset,
          unknownset,result;
     incres:=incmat(G);
     incmatrix:=incres[1];
     incmatcol:=incres[2];
     #incmatcol is a recorder to record the order of the columns
     #of the incidence matrix. It is used for later multiplication
     #with the currents.

     dirtbset:=ends(T);
     #dirtbset is the branches set for the tree.

     NTB:=nops(dirtbset);
     unknowni:=[seq(0,i=1..NTB)];
     #unknowi is a list used to store the currents of the tree branches.
     #These currents will be treated as the unknowns in the kcl equat-
     #ions.
     #NOTE: At this stage, the current of the voltage sources VEH are
     #also treated as unknowns.

     counter:=0;

     NE:=nops(incmatcol);
     ivector:=Vector(NE);
     #ivector is used to store all the current variables.

     for k from 1 to NE do
          te:=edges(incmatcol[k],G);
          tw:=eweight(op(te),G);
          neww:=cat('i',tw);
          ivector[k]:=neww;
          #form the current variables.

          if member(incmatcol[k],dirtbset)=true then
               counter:=counter+1;
               unknowni[counter]:=neww;
               #if the branch is in the tree, then the current
               #variable is treated as the unknowns.
          end if;
     end do;

     res:=incmatrix.ivector;
     #get all the kcl equations for all the nodes.
     M:=Dimension(res);
     elist:=[seq(0,i=1..M)];
     for i from 1 to M do
          elist[i]:=res[i];
     end do;
     #get the equation set for later use of solve.

     eqset:=convert(elist,set);
     unknownset:=convert(unknowni,set);

     result:=solve(eqset,unknownset);
     #express the unknown currents in terms of the known currents.
     #That is to say, express the tree currents in term of the
     #cotree currents.
     #NOTE: There is no guarantee that the resulted kcl equation will
```

```
    #have the current sources JFG at the right hand side of these
    #equations. It they are not, then they will not appear in the
    #compact SFG later generated.

    return(result);
end proc;

#kvl calls a function named "mykvl" which calculates a
#single KVL equation for the inputed branch and the weight.

#the inputs to this function is the tree of the diagram and the
#cotree of the diagram.
#the output of the function is a set of KVL equations.

#for the final returned values of this function, see the head of the
#comments of the function "mykvl".

kvl:=proc(Tree::graph,CoTree::graph)
    local bsetCT,NCT,eqlist,i,CTbranch,CTedge,CTweight,
        reseq;

    bsetCT:=ends(CoTree);
    #get all the cotree links for forming loop at later stage.

    NCT:=nops(bsetCT); eqlist:={};
    #eqlist is used to store the set of the equations which
    #express the cotree voltages in terms of tree voltages.
    #NOTE: at this stage, the voltages of the current sources
    #are expressed in terms of tree branches voltages.
    for i from 1 to NCT do
        CTbranch:=bsetCT[i];
        CTedge:=edges(CTbranch,CoTree);
        CTweight:=eweight(op(CTedge),CoTree);
        #get the branch and the weight of the cotree link
        #which is to be calculated on the KVl equations.

        reseq:=mykvl(Tree,CTbranch,CTweight);
        #for the result of the mykvl, see the comments at the
        #head of function "mykvl".

        eqlist:=eqlist union reseq;
    end do;

    return (eqlist);
end proc;

#miniohm generates the ohm's law according to the input. The input
#must be in the form as VR12 or iL12. Not in V12,i34.

miniohm:=proc(ELE)
    local itype,element,etype,viele;
    itype:=substring(ELE,1);
    #the type of the input, either i or V.

    element:=substring(ELE,2..length(ELE));
    #the element needs the Ohm's law. This is the reason that the
    #input must be VR12. R12 is needed to determine the element.

    etype:=substring(element,1);
    #the type of the element.

    if itype='i' then
    #if the input type is i then output i=..V.

        viele:=cat('V',element);
        if etype='R' then
            return(ELE=viele/element);
        elif etype='C' then
            return(ELE=S*element*viele);
        elif etype='L' then
            return(ELE=viele/(S*element));
        end if;
    elif itype='V' then
    #if the input type is V, then output is V=..i.

        viele:=cat('i',element);
        if etype='R' then
            return(ELE=element*viele);
        elif etype='C' then
            return(ELE=1/(S*element)*viele);
        elif etype='L' then
            return(ELE=S*element*viele);
        end if;
    end if;
end proc;

#mycdc outputs a number which is the highest n-th order of loops in the
#graph and a list whose entries are the sum of the n loop weights, where
#n is the index of the entry. Say, res[1] is the sum of the 1st loop
```

```
#weights.

mycdc:=proc(G::graph)
     local wofb,sumlw,res,N1,i,L,M,N1loop,maxlen,j,temp,k,
          Ne1loop,l,lnum,lw,weightk;
     sumlw[1]:=0;
     #a table used to store the values for the sum of the n-th loop
     #weights in the n-th entries of the table. Say, sumlw[1] is the
     #sum of the 1st loop weights of the graph and sumlw[2] is the
     #sum of the 2nd loops of the graph.

     res:=w1loop(G);
     #res is an array whose entries are the loop weights for the
     #corresponding index 1st loop. Say, res[1] is the loop weight
     #for the first 1st loop.

     N1:=nops(convert(res,list));
     #simply convert the array to a list and then get the total
     #numbers of the operands in the list, i.e. get the 1st loop number.

#if there is no 1st loops for the given graph, then the sums of all the
#loop weights of the graph are 0.
     if N1=0 then
          return(0,[]);
     end if;

#else there is 1st loops then calculate the sum of the 1st loop weights
#for the graph.
     for i from 1 to N1 do
          sumlw[1]:=sumlw[1]+res[i];
          end do;


          L:=fnloop(G);
          #find the n-th loops of the given graph.


          M:=L[1];
     #If there is no 2nd loops for the given graph, then the highest order
     #of the given graph will be 1 and also return the sum of 1st loop
     #weights.
          if M=0 then
               return(1,[sumlw[1]]);
          end if;


     #else if there are 2nd loops.


          N1loop:=nops(convert(L[2],list));
     #L[2] is the array which contains the number of branches for the
     #corresponding indices. Say, L[2][1] is the number of branches for
     #node 1 and L[2][4] is the number of branches for node 4. For the
     #meaning of branches, see Fig14-11 on page 555 of [6].
     #N1loop is the total number of vertices or nodes for the graph.


          maxlen:=1;
          for i from 1 to N1loop do
          #traverse each nodes.


               for j from 1 to L[2][i] do
               #traverse each branches for each nodes.


                    temp:=nops(L[3][i][j])-1;
```

```
                #found out the length of the branch L[3][i][j].
                #L[3][i][j] is the j-th branch for the i-th node.
                if temp > maxlen then

                     maxlen:=temp;
                     #found out the maxium lenght of all of the branches.
                     #This number is the number of the highest order loops
                     #for the graph.
                end if;
           end do;
      end do;

#this loop extract the nodes for the 2nd to high order loops. For a
#detailed description, refer to [6].
      for k from 2 to maxlen do
           sumlw[k]:=0;
           #k is the index for the k-th loop, it is used to extract the
           #first k-th nodes from all the branches.

           for i from 1 to N1loop do
           #Remember, N1loop is the total number of all the vertices or
           #nodes in the graph.

                Ne1loop:=L[2][i];
                #Ne1loop is the total number of the branches for node i.

                for j from 1 to Ne1loop do
                #traverse each branches of node i.

                     weightk:=1;
```

```
#the initial value for the product of the k-th loop
#weights.

for l from 1 to k do
#extract the first k nodes from the branches. For a
#detailed reason to do this, see page 555 of [6].

    if k<= nops(L[3][i][j])-1 then
    #if the L[3][i][j] branch still have k nodes.
    #L[3][i][j] is the j-th branch for the i-th
    #node.

        lnum:=L[3][i][j][l];
        #L[3][i][j][l] is the l-th entries of
        #branch L[3][i][j]. This entry is actually
        #the index number of the 1st loops.
        #For detailed explanation, see page 555
        #of [6].

        if lnum <> 0 then
            lw:=res[lnum];
            #get the 1st-loop-weight for this loop.

            weightk:=weightk*lw;
        end if;
    end if;
end do;
sumlw[k]:=sumlw[k]+weightk;
#calculate the high order loop weights from this
#1st loop weights.
```

```
                end do;
            end do;
        end do;
        return (maxlen,ind2l(op(eval(sumlw))));
        #maxlen is the number of the highest order loops in the graph and
        #the list is the sum of the n-th loop weights, where n is the index
        #of the entry.
    end proc;


    #mykvl applies the KVL for a specific cotree branch. It reads
    #in a Tree, a cotree branch, and the weight of the cotree branch.
    #It returns an equation which left hand side is the cotree link's
    #voltage and the rhs is the tree branches voltages.


    #ATTENTION: when forming the loop to use the KVL equation to express the
    #voltages of the cotree links, there is no way to guarantee that the
    #voltage sources (VEH) will be included because there is no way to
    #guarantee that the loop will be formed in the way that the tree branches
    #whose weight is the voltage sources must be in the loop to form the KVL
    #equation.
    #NOTE: The result of the above thing is that if the voltage sources are
    #not included in the KVL equations which are returned by this function,
    #then these voltage sources will not appear in the compact SFT at the
    #later time.


    mykvl:=proc(T::graph,CotreeBranch::list,WeightCB)
        local OrigTree,Untree,L,reloop,direloop,M,eqlist,j,tb,origedge,
            origw,gredge,tpw,neww,dire,i,result;
        OrigTree:=duplicate(T);
        #because it needs to add an edge to the tree, it needs to duplicate
```

```
     #a new tree.

     addedge(CotreeBranch,weights=WeightCB,OrigTree);
     #add the cotree link with its weights to form a loop for KVL
     #equation.

     Untree:=dig2udig(OrigTree);
     #the loops for the KVL equation should be a loop in the undirected
     #graph.

     L:=fpath(Untree,CotreeBranch[1],CotreeBranch[2]);
     #to find the path between the cotree nodes is equivalent to find
     #the loop of them.

#ATTENTION: in directed graph, two nodes can form a loop. this loop in
#the undirected graph will become a single path. in undirected graph,
#at least three nodes needs to form a loop.

     if nops(L[2])=1 then
          reloop:=L[2][1];
     else
          if nops(L[2][1])>2 then
               reloop:=L[2][1];
          else
               reloop:=L[2][2];
          end if;
     end if;
     #Note, there are only two situations in fpath. One is fpath
     #returns only one path. This path in directed graph is a loop.
     #The other situation is fpath returns exactly two paths.
```

```
#The path with more than two nodes is the path needed.

reloop:=[op(reloop),CotreeBranch[1]];
#construct the loop with the first node of CotreeBranch

direloop:=rsort(reloop);
#reverse the orders of reloop to form direloop which has the
#positive direction of the loop.

M:=nops(direloop);
#M nodes in the loop. Therefore, M-1 branches or weights in the
#equation

eqlist:=[seq(0,i=1..M-1)];
#to store the items in the final single KVL equation.

for j from 1 to M-1 do
     tb:=[direloop[j],direloop[j+1]];
     origedge:=edges(tb,OrigTree);
     #when extract the edge number of tb, there is a possibility
     #that the direction is the same or opposite.

     if j<>1 then
          if nops(origedge)>0 then
               origw:=eweight(op(origedge),OrigTree);
               neww:=cat('V',origw);
               neww:=-neww;
               #when the branches of the directed
               #tree has the same direction with the
               #directed loop, the weight of the tree
```

```
                #branch should be negative in the KVL
                #equation because it is at the rhs of the
                #equation. Otherwise, it is positive.

            else
                tb:=rsort(tb);
                #reverse the order of the branch
                gredge:=edges(tb,OrigTree);
                origw:=eweight(op(gredge),OrigTree);

                neww:=cat('V',origw);
                neww:=neww;
                #the branches of the directed tree here
                #has the opposite direction with the directed
                #loop, so their weights is positive in KVL
                #equation.

            end if;
        else

            origw:=eweight(op(origedge),OrigTree);
            neww:=cat('V',origw);
            #for the first two nodes in the directed loop,
            #there is no need to judge their direction.
            #because these two nodes are the cotree link nodes and
            #the direction of the directed loop is set according
            #to their direction.
        end if;
        eqlist[j]:=neww;
    end do;
    result:={eqlist[1]=convert([seq(eqlist[i],i=2..nops(eqlist))],'+')};
    #convert the variables in eqlist into a form of equation.

    return(result);
end proc;

#ohm:
#input: a graph
#output: generate the Ohm's Law for the RLC elements. For the elements
#of VEH and JFG, there surely will be no Ohm's law for these kinds of
#elements.

#The input of this function is a circuit diagram generated by "ccgraph",
#and the tree and the cotree of the diagram, which were generated by
#"ctree".
#The output of this function is two sets of equations, one is for the
#tree voltages, the other is for the cotree currents. The sources VEH
#and JFG are not included in the Ohm's equation returned by this function.

ohm:=proc(G::symbol,Tree::graph,CoTree::graph)
    local eqlistT,eqlistCT,bsetT,NT,k,Tbranch,Tedge,
        Tweight,weighttype,weightsuffix,lhseq,rhseq,eq,bsetCT,NCT,
        CTbranch,CTedge,CTweight;
    eqlistT:={}; eqlistCT:={};
    #initialize the two sets of the equations.

    bsetT:=ends(Tree);
    #bset is the branches set for the tree of the circuit diagram
```

```
NT:=nops(bsetT);

for k from 1 to NT do
#for each branches in the tree, do the following works. Note,
#at present time, the VEH are under consideration.

    Tbranch:=bsetT[k];
    Tedge:=edges(Tbranch,G);
    Tweight:=eweight(op(Tedge),G);
    #get the weights of each of the branches of the tree.

    weighttype:=substring(Tweight,1);
    #get the type of the weights, i.e. the type of the
    #elements.

    lhseq:=cat('V',Tweight);
    rhseq:=cat('i',Tweight);
    #because it is in the tree, so, the lhs of the Ohm's
    #Law is Vsomething and the rhs of the Ohm's Law is
    #isomething. Note, at this stage, no consideration
    #on VEH and JFG are made. That is to say, even for
    #the sources, the two sides of the Ohm's Law are also
    #formed at this stage.

    if weighttype='R' then
        eq:={lhseq=Tweight*rhseq};
        eqlistT:=eqlistT union eq;
    elif weighttype='C' then
        eq:={lhseq=rhseq/(S*Tweight)};
        eqlistT:=eqlistT union eq;
```

```
            elif weighttype='L' then
                 eq:={lhseq=S*Tweight*rhseq};
                 eqlistT:=eqlistT union eq;
            end if;
            #at this point, only the Ohm's Law for RLC are generated,
            #and the VEH JFG sources are ignored. Nothing is done on
            #them. As the result, the returned Ohm's Law of this
            #function will not have VEH and JFG in the returned
            #equations.

       end do;

  #for cotree elements, it is similar with the tree elements.
       bsetCT:=ends(CoTree);
       NCT:=nops(bsetCT);

       for k from 1 to NCT do
            CTbranch:=bsetCT[k];
            CTedge:=edges(CTbranch,G);
            CTweight:=eweight(op(CTedge),G);
            weighttype:=substring(CTweight,1);
            lhseq:=cat('i',CTweight);
            rhseq:=cat('V',CTweight);
            #because it is in the cotree, the lhs of Ohm's Law is
            #Isomething and the rhs of the Ohm's Law is Vsomething.
            #at this stage, the two sides for the sources' Ohm's
            #Law are also generated, although they are meaningless.

            if weighttype='R' then
                 eq:={lhseq=rhseq/CTweight};
```

```
                eqlistCT:=eqlistCT union eq;
           elif weighttype='C' then
                eq:={lhseq=S*CTweight*rhseq};
                eqlistCT:=eqlistCT union eq;
           elif weighttype='L' then
                eq:={lhseq=rhseq/(S*CTweight)};
                eqlistCT:=eqlistCT union eq;
           end if;
           #generate the Ohm's Law for the RLC and leave VEH and
           #JFG unchanged. The returned cotree current equations
           #will also not have current sources.

       end do;

      return(eqlistT,eqlistCT);
end proc;

#the input to parsinp is Compact SFG, the input variable and the
#output variable, between which is to be calculated the transfer
#function.

#this function parses the input of the SFG analysis. It extends the
#limitation on the inputs. The inputs can be any linear combinations
#of V and i variables. If the V is not in the inputed CSFG (Compacted
#SFG), then Ohm's Law is used. It is the same with the i's case. If
#the i is not in the cotree, the ohm's law is also used. Then it adds
#some new vertices and branches and weights if necessary.

#if the user input the voltage of voltage sources or the current of
#current sources but the sources are not in the CSFG, then these sources
```

```
#with the prefix "V" or "i" are added to CSFG and make the necessary
#steps to ensure the resulted closed CSFG is connected.


#THE CASES THIS FUNCTION CAN NOT HANDLE AT THIS STAGE:
#       EITHER OF THE INPUT IS 2*Vxx OR a*Vxx, I.E. SOMTHING
#       TIMES THE VARIABLE. THIS LEADS TO THE CONSEQUENCE THAT
#       THE INPUT CAN NOT BE V12+V12 EITHER BECAUSE MAPLE TREATS
#       IT AS 2*V12.
#       IN THIS CASE AN ERROR MESSAGE ABOUT THE SUBSTRING FUNCTION
#       WILL BE GENERATED BECAUSE THE SUBSTRING FUNCTION CAN NOT HANDLE
#       THE SUBSTRING OF A NUMBER OR A `^`TYPE.

parsinp:=proc(CSFG::graph,INI,OUT)
    local RetCSFG,vset,NINI,NOUT,incc,ohmeq,L,newv,newe,neww,DINI,
        j,tpitem,eq,res,NV,k,outcc,DOUT,tpres,RCCSFG,initype,
        itemtype,outtype,tmp;
    RetCSFG:=duplicate(CSFG);
    #duplicate the original CSFG (Compact SFG).
    vset:=vertices(CSFG);
    NINI:=nops(INI);
    NOUT:=nops(OUT);

    if evalb(NINI=1)=true then
        if has(vset,INI)=true then
            incc:=INI;
        #if input is only one variable and also it is in the vset,
        #then it is surely the incc (input to coding of csfg).

        else
```

```
initype:=substring(INI,1..2);
if initype='VV' or initype='VE'
     or initype='VH' or initype='IJ'
     or initype='IF' or initype='IG' then
     newv:=INI;
     addvertex(newv,RetCSFG);
     incc:=INI;
#if the inputed type is voltage sources or
#current sources but they are not in the CSFG,
#then they are added to the CSFG and make them
#as the head node of the closed CSFG.

elif initype='iV' or initype='iH'
     or initype='iE' or initype='VJ'
     or initype='VF' or initype='VG' then
     error "please reinput the input varible";
#if the input is the current of voltage source or
#voltage of current source, then you need to change it.
else
#otherwise, use Ohm's Law to convert the input
#to the I and V equivalence.

     ohmeq:=miniohm(INI);
     #miniohm is a small code to implement Ohm's Law.

     L:=extractvw(ohmeq);
#extractvw is another small code to extract the operands
#and the coefficients.

     newv:=INI;
```

```
                newe:=[op(L[2]),INI];
                neww:=op(L[3]);
                addvertex(newv,RetCSFG);
                addedge(newe,weights=neww,RetCSFG);
                incc:=INI;
            #If it is not in the vset, then ohm's law is used and
            #also add some vertices and branches and weights if
            #necessary.

            end if;
        end if;
else
#if there are more than one operands in the input.

        DINI:=INI;
        #due to the later substitution, it is better to duplicate
        #another equation.

        for j from 1 to NINI do
            tpitem:=op(j,INI);

            #here should add some guarding mechanism to guarantee
            #that if user input Vxx+2*Vxx, there is some way to
            #discriminate 2 from the V of Vxx. Otherwise, the
            #substring function will failed. At this stage,
            #no way to watch it. So we can not give input and output
            #variables as the form:Vxx+n*Vxx.

            if has(vset,tpitem)=false then
                itemtype:=substring(tpitem,1..2);
                if itemtype='VV' or itemtype='VE'
                     or itemtype='VH'
                     or itemtype='IJ' or itemtype='IF' or
                     itemtype='IG' then
                     addvertex(tpitem,RetCSFG);
                     DINI:=DINI;
           #ALSO, IT SHOULD JUDGE WHETHER THIS OPERAND IS
           #A SOURCE OR NOT, IF YES, THEN SIMPLY ADD IT AND
           #LEAVE THE DINI UNCHANGED. IF NOT, THEN THE
           #FOLLOWING STEPS ARE TAKEN.
                elif itemtype='iV' or itemtype='iE'
                     or itemtype='iH' or itemtype='VJ'
                     or itemtype='VF' or itemtype='VG' then
                     error "please reinput the input varible";
           #if the input is the current of voltage source or
           #voltage of current source, then you need to change it.
                else
                         ohmeq:=miniohm(tpitem);
                     DINI:=expand(subs(ohmeq,DINI));
                end if;
           end if;
      end do;
#for each of the operands in the input, if it is not in the vset,
#then ohm's law is used and then make the corresponding changes
#on the inputs.

      newv:='Vin';
      #just introduce a new vertex. It can be meaningless thing.
      addvertex({newv},RetCSFG);
```

```
          eq:=newv=DINI;
          res:=extractvw(eq);
          #extract the coefficients and the operands.

          NV:=nops(res[2]);
          for k from 1 to NV do
               newe:=[res[2][k],newv];
               neww:=res[3][k];
               addedge(newe,weights=neww,RetCSFG);
          end do;
          #add the new edges and the new weights.

          incc:=newv;
          #make the new node as the initial node for the usage of
          #generating a closed CSFG.
     end if;

     if evalb(NOUT=1)=true then
     #It is similar with the judgments made on INI.

          if has(vset,OUT)=true then
               outcc:=OUT;
          else
               outtype:=substring(OUT,1..2);
               if outtype='VV' or outtype='VE'
                    or outtype='VH' or outtype='IJ'
                    or outtype='IF' or outtype='IG' then
                    newv:=OUT;
                    addvertex(newv,RetCSFG);
                    outcc:=OUT;
                    #JUDGE WHETHER THIS OPERAND IS THE SOURCE OR NOT,
                    #IF YES, THEN ADD IT AND MAKE IT AS THE TAIL OF
                    #THE CLOSED CSFG. IF NOT, THEN OHM'S LAW IS USED.
                    elif outtype='iV' or outtype='iE'
                            or outtype='iH' or outtype='VJ'
                            or outtype='VF' or outtype='VG' then
                            error "please reinput the output varible";
                    #if the output is the current of voltage source or
                    #voltage of current source, then you need to change it.
                    else
                        ohmeq:=miniohm(OUT);
                        L:=extractvw(ohmeq);
                        newv:=OUT;
                        newe:=[op(L[2]),OUT];
                        neww:=op(L[3]);
                        addvertex(newv,RetCSFG);
                        addedge(newe,weights=neww,RetCSFG);
                        outcc:=OUT;
                    end if;
                end if;
            else
                DOUT:=OUT;
                for j from 1 to NOUT do
                    tpitem:=op(j,OUT);

                    #here should add some guarding mechanism to guarantee
                    #that if user input Vxx+2*Vxx, there is some way to
                    #discriminate 2 from the V of Vxx. Otherwise, the
                    #substring function will failed. At this stage,
                    #no way to watch it.
```

```
        if has(vset,tpitem)=false then
            itemtype:=substring(tpitem,1..2);
            if itemtype='VV' or itemtype='VE'
                or itemtype='VH' or itemtype='IJ'
                or itemtype='IF' or itemtype='IG' then
                addvertex(tpitem,RetCSFG);
                DOUT:=DOUT;
        #JUDGE WHETHER IT IS A SOURCE OR NOT, IF YES, THEN
        #ADD IT AND DOUT LEAVE UNCHANGED. IF NOT THEN OHM'S
        #LAW IS USED.
            elif itemtype='iV' or itemtype='iE'
                or itemtype='iH' or itemtype='VJ'
                or itemtype='VF' or itemtype='VG' then
                error "please reinput the output varible";
        #if the output is the current of voltage source or
        #voltage of current source, then you need to change it.

            else
                ohmeq:=miniohm(tpitem);
                DOUT:=expand(subs(ohmeq,DOUT));
            end if;
        end if;
    end do;

    newv:='Vout';
    addvertex({newv},RetCSFG);

    eq:=newv=DOUT;
    res:=extractvw(eq);
```

```
          NV:=nops(res[2]);
          for k from 1 to NV do
               newe:=[res[2][k],newv];
               neww:=res[3][k];
               addedge(newe,weights=neww,RetCSFG);
          end do;
          outcc:=newv;
     end if;

     tpres:=codingsfg(RetCSFG,incc,outcc);
     #according to the above results, coding the CSFG.

     RCCSFG:=tpres[1];
     #RCCSFG is for "Returned Closed Compact SFG".
     return(RCCSFG);
end proc;

#The first argument to rop is the list which will be search for
#the second inputed argument, num. When num is found in the list, the
#index of num in the list is returned. If num is not in the list, then
#FAIL is returned.

rop:=proc(L::list,num)
     local N,k;
     N:=nops(L);
     for k from 1 to N do
          if L[k]=num then
          return(k);
          end if;
          end do;
          return(FAIL);
     end proc;

     #rsort must  accept an ascending numerical ordered list which was
     #produced by the Maple "sort" command for a list. This program then
     #swaps the corresponding entries in the ordered list to achieve the
     #reversel ordered list. Therefore, this function is actually swapping.
     #but its function here is not only swapping, that is why I did not use
     #sort in maple.

     rsort:=proc(L::list)
          local i,k,temp,target,len,res;
          k:=nops(args);
          res:=args;
          if (k mod 2)=0 then
               len:=k/2;
               for i from 1 to len do
                    target:=k+1-i;
                    temp:=res[i];
                    res[i]:=res[target];
                    res[target]:=temp;
               end do;
          else
               len:=(k-1)/2;
               for i from 1 to len do
                    target:=k+1-i;
                    temp:=res[i];
                    res[i]:=res[target];
                    res[target]:=temp;
          end do;
     end if;
return(res)
end proc;

#sfgmain is the main function for the implementation of the SFG analysis.
#It call several other packages.

#the input to this function is a compact SFG obtained from the function
#called "gencsfg". The second input argument is the input variable
#in the transfer function and the third argument is the output variable
#in the transfer function.

#the output of the function is an symbolic expression of the transfer
#function.

sfgmain:=proc(ComSFG::graph,input,output)
     local ccsfg,deltac,Zcoeff,tranfnum,tranfdenom;
     ccsfg:=parsinp(ComSFG,input,output);
     #parse the inputs and the outputs from the user and then
     #generate the closed SFG according to these two arguments.

     deltac:=cdc(ccsfg);
     #calculates the Deltac of the closed SFG.

     Zcoeff:=coeff(deltac,Z);
     #extract the coefficients of Z. It is F in [6].

     tranfnum:=Zcoeff;
     tranfdenom:=simplify(deltac-Zcoeff*Z);
     #get the transfer function

     return(normal(tranfnum/tranfdenom));
end proc;

#w1loop calculates the weights of the 1st loops of the given
#graph. It returns an array whose entries are the weights of the 1st
#order loops corresponding to the indices of the array.
#it calls f1loop and wbranch.

w1loop:=proc(G::graph)
     local wofb,L,N,wof1loop,i,M,j,current,nnode;
     wofb:=wbranch(G);
     #wofb is a matrix "Weight OF Branches". It is the matrix used
     #to store the branch weights of the given graph G.
     L:=f1loop(G);
     N:=L[1];
     #get the number of the 1st loops in the graph and also the
     #loops themselves.

     if N=0 then
          return ([]);
     end if;
     #if there is no 1st loops in the given graph, then empty list
     #is returned.

     wof1loop:=array(1..N,[seq(1,i=1..N)]);
     #the array used to store the 1st-loop weights of the graph.
     #It can also use list instead of array here. But if the total
     #number of the 1st loop in the graph is too large, there will
```

```
    #be an error from Maple said "ERROR: trying to assign values to
    #a long list.
    for i from 1 to N do
        M:=nops(L[2][i]);
        for j from 1 to M-1 do
            #this is used to calculate the loop weight of each 1st
            #loops found before.
            current:=op(j,L[2][i]);
            nnode:=op(j+1,L[2][i]);
            #to get the ends of the branches. [current,nnode] is the
            #directed branch in the loop.
            wof1loop[i]:=wof1loop[i]*wofb[current,nnode];
            #multiply the branch weights to form the loop weight.
        end do;
    end do;

    return(eval(wof1loop));
end proc;

#wbranch accepts a graph and then outputs a matrix whose entries
#are the weights of the branches of the inputed graph and the row number
#is the tail node number and the column number is the head node number.

#Due to the fact that this program uses the concepts of head and 'Out'
#in the 'incident' command, this program is strongly limited by the
#graph. The inputed graph to this program must be one direction graph.
#As the result, undirected graph and two directions graph such as
#petersen is not valid to this program.
wbranch:=proc(G::graph)
    local N,weight,i,outedge,outhead,M,j,edgeweight,headnode;
    N:=nops(vertices(G));
    #the total number of vertices in the graph. NOTE that these vertices
    #are supposed to be continuous numbers, such as 1,2,3...N .  If the
    #numbers are not continuous, then this program will fail on this
    #case.
    weight:=Matrix(N,N);
    #the matrix used to store the weights of the branches. It is a
    #square matrix whose row and column represents the nodes of the
    #graph.
    for i from 1 to N do
        outedge:=convert(incident(i,G,Out),list);
        outhead:=head(outedge,G);
        #get the possible column numbers of the given row.
        M:=nops(outedge);
        for j from 1 to M do
            edgeweight:=eweight(op(j,outedge),G);
            headnode:=outhead[j];
            weight[i,headnode]:=edgeweight;
            #add each weight to the corresponding matrix element.
        end do;
    end do;
#Because of the fact that for the circuits which I deal with in my
#thesis, the number of nodes is not very large, therefore, the weight
#results are stored in a matrix in this program. Even for petersen(),
#there are only 10 nodes. But for a large amount number of nodes and
#loosely connected circuits, this matrix has better be changed into
#three arrays to represent its sparse matrix .
    return (evalm(weight));
end proc;
```

```
#pay attention to the help file of eweight that changing the edge
#weight table will directly affect the graph. This might be used
#in the function which changes the voltage source weight in a
#graph.

#weight2b itself reads in a directed graph and the edge weight
#which is needed to search in the graph. If the weight is found
#in the graph, then the vertex pair of the edge whose weight is the
#searched weight are returned as a set of lists. If the weight is not
#in the graph, then FAIL is returned.

#for searching the branches having voltage sources, independent
#or dependent, the the weight needs to be pre-processed by extracting
#the first char of the weight.

weight2b:=proc(G::graph,wei)
     local L,indarray,entarray,counter,resbranch,N,i,rese,
          M,tailnode,headnode;
     L:=eweight(G);
     indarray:=[indices(L)];
     entarray:=[entries(L)];
     counter:=0;

     N:=nops(indarray);
     for i from 1 to N do
          if op(op(i,entarray))=wei then
               counter:=counter+1;
               rese[counter]:=op(op(i,indarray));
          end if;
     end do;
```

```
        #find the edges of corresponding weight.
        if counter=0 then
             return(FAIL);
        end if;

        M:=counter;
        for i from 1 to M do
             tailnode:=tail(rese[i],G);
             headnode:=head(rese[i],G);
             resbranch[i]:=[tailnode,headnode];
        end do;
        #got the tail node and head node of those edges.
        return(M,ind2l(op(eval(resbranch))));
   end proc;

   end module;
```

Appendix B

Source Code for mna

```
mna:=module()
     export mnamain;
     local dump, ind2l, rop;

#this function dumps out the inputed values for the later usage.
#it calls another program "ind2l",which covert a table to ordered list.
#user input syntax: beginput, elementname,pos,neg, (controlpos,
#controlneg,control-coefficient, varible),sp,elementname,pos,neg,
#(controlpos controlneg,control-coefficient,varible), sp,endinput.

dump:=proc()
     local gnum,enum,i,inp,totale;
     if args[1]<> begininput or args[nargs]<> endinput then
          error("syntax error");
     end if;

     gnum:=1;enum:=1;
     for i from 2 to nargs-1 do
          if args[i] <> sp then
               inp[gnum][enum]:=args[i];
               enum:=enum+1;
          else
```

```
                    gnum:=gnum+1;
                    enum:=1;
               end if;
          end do;

          totale:=gnum;

          return(totale,[seq(ind2l(op(eval(inp[i]))),i=1..gnum)]);

     end proc;

     #this program calls another program "rop", which find the position of a
     #given number in a list.

     #the input of this function must be op(eval(T)), suppose that T is a one
     #level table.

     ind2l:=proc(LL::list)
          local counter,N,temp1,temp2,k,trhs,result,position,i;
          counter:=0;
          N:=nops(LL);
          temp1:=[seq(-1,i=1..N)];
          temp2:=[seq(-1,i=1..N)];
          for k from 1 to N do
               temp1[k]:=lhs(LL[k]);
               trhs:=rhs(LL[k]);
               if (trhs=NULL) then
                    temp2[k]:=-infinity;
                    counter:=counter+1;
               else
```

```
            temp2[k]:=trhs;
        end if;
    end do;

    result:=[seq(-1,i=1..N-counter)];
    for k from 1 to N-counter do
        position:=rop(temp1,k);
        result[k]:=temp2[position];
    end do;
    return (result);
end proc;

#The first argument to this function is the list which will be search
#for the second inputed argument, num. When num is found in the list,
#the index of num in the list is returned. If num is not in the list,
#then FAIL is returned.

rop:=proc(L::list,num)
    local N,k;
    N:=nops(L);
    for k from 1 to N do
        if L[k]=num then
        return (k);
        end if;
    end do;
    return(FAIL);
end proc;

mnamain:=proc()
    local l,nod,counter,temp,i,ty,tnum,n,A,B,vseq,cvseq,X,k,mval,typ,
     Y,cof,cb,ce,solres,N,eq,t,h,nn;
l:=dump(args);
   nod:={};
counter:=0;
temp:=1;
for i from 1 to l[1] do
     nod:={l[2][i][2],l[2][i][3]} union nod;
     ty:=substring(l[2][i][1],1);
          if ty=V or ty=E or ty=H then
               counter:=counter+1;
          end if;
end do;
tnum:=nops(nod);
n:=tnum+counter;
A:=Matrix(n,n);
B:=Matrix(n,1);
vseq:=seq(cat(V,i),i=1..tnum);
vseq:=convert([vseq], Matrix);
cvseq:=Transpose(vseq);
X:=Matrix(n,1,[cvseq]);
for k from 1 to l[1] do
     mval:=l[2][k][1];
     typ:=substring(mval,1);
     t:=l[2][k][2];
     h:=l[2][k][3];
     if typ=R  then
          A[t,t]:=1/mval+A[t,t];
          A[t,h]:=-1/mval+A[t,h];
          A[h,t]:=-1/mval+A[h,t];
          A[h,h]:=1/mval+A[h,h];
```

```
elif typ=C then
     A[t,t]:=s*mval+A[t,t];
     A[t,h]:=-s*mval+A[t,h];
     A[h,t]:=-s*mval+A[h,t];
     A[h,h]:=s*mval+A[h,h];
elif typ=L then
     A[t,t]:=1/(s*mval)+A[t,t];
     A[t,h]:=-1/(s*mval)+A[t,h];
     A[h,t]:=-1/(s*mval)+A[h,t];
     A[h,h]:=1/(s*mval)+A[h,h];
elif typ=J then
     B[t,1]:=-mval+B[t,1];
     B[h,1]:=mval+B[h,1];
elif typ=G then
       cof:=l[2][k][6];
     cb:=l[2][k][4];
     ce:=l[2][k][5];
     A[t,cb]:=cof+A[t,cb];
     A[t,ce]:=-cof+A[t,ce];
     A[h,cb]:=-cof+A[h,cb];
     A[h,ce]:=cof+A[h,ce];
elif typ=V then
     nn:=tnum+temp;
     A[t,nn]:=1+A[t,nn];
     A[h,nn]:=-1+A[h,nn];
     A[nn,t]:=1+A[nn,t];
     A[nn,h]:=-1+A[nn,h];
     B[nn,1]:=mval+B[nn,1];
     X[nn,1]:=cat('i',mval);
     temp:=temp+1;
```

```
elif typ=E then
     Y:=l[2][k][7];
     cof:=l[2][k][6];
     nn:=tnum+temp;
     cb:=l[2][k][4];
     ce:=l[2][k][5];
     A[nn,t]:=1+A[nn,t];
     A[nn,h]:=-1+A[nn,h];
     A[nn,cb]:=-cof+A[nn,cb];
     A[nn,ce]:=cof+A[nn,ce];
     A[t,nn]:=1+A[t,nn];
     A[h,nn]:=-1+A[h,nn];
     X[nn,1]:=cat('i',mval);
     temp:=temp+1;
elif typ=H then
     Y:=l[2][k][7];
     cof:=l[2][k][6];
     nn:=tnum+temp;
     cb:=l[2][k][4];
     ce:=l[2][k][5];
     A[nn,t]:=1+A[nn,t];
     A[nn,h]:=-1+A[nn,h];
     A[nn,cb]:=-cof*Y+A[nn,cb];
     A[nn,ce]:=cof*Y+A[nn,ce];
     A[t,nn]:=1+A[t,nn];
     A[h,nn]:=-1+A[h,nn];
     X[nn,1]:=cat('i',mval);
     temp:=temp+1;
elif typ=F then
     Y:=l[2][k][7];
```

```
                    cof:=l[2][k][6];
                    nn:=tnum+temp;
                    cb:=l[2][k][4];
                    ce:=l[2][k][5];
                    A[t,cb]:=cof*Y+A[t,cb];
                    A[t,ce]:=-cof*Y+A[t,ce];
                    A[h,cb]:=-cof*Y+A[h,cb];
                    A[h,ce]:=cof*Y+A[h,ce];
                    temp:=temp+1;

               end if;
          end do;
          A:=DeleteRow(A,tnum..tnum);
          A:=DeleteColumn(A,tnum..tnum);
          B:=DeleteRow(B,tnum..tnum);
          X:=DeleteRow(X,tnum..tnum);

          solres:=LinearSolve(A,B);
          N:=RowDimension(solres);
          for i from 1 to N do
               eq[i]:=X[i,1]=solres[i,1];
          end do;

          return(ind2l(op(eval(eq))));
     end proc;
     end module;
```

REFERENCES

[1] Martha L. Abell and James Braselton. *Maple V by example.* AP Professional, Boston, 1994.

[2] A.Konczykowska, M.Bon, H.Wang, and R.Mezui-Mintsa. Symbolic analysis as a tool for circuit optimization. *IEEE, Int. Symp. Circuits Syst.*, May 1992.

[3] Water Banzhaf. *Computer-Aided Circuit Analysis Using Spice.* Prentice-Hall, New Jersey, 1989.

[4] Joseph Carr. *Op-amp Circuit Design and Applications.* Blue Ridge Summit, Pa: G/L Tab Books, 1976.

[5] Sin-Min Chang. *Symbolic Analog Circuit Analysis.* PhD thesis, Michigan State University, 1992.

[6] Leon O. Chua and Pen-Min Lin. *Computer-Aided Analysis of Electronic Circuits: Algorithms and Computational Techniques.* Englewood Cliffs, N.J.:Prentice-Hall, 1975.

[7] Robert M. Corless. *Essential Maple: an introduction for scientific programmers.* Springer-Verlag, New York, 1995.

[8] Thomas A. DeMassa. *Electrical and Electronic Devices, Circuits, and Instruments.* West Publishing Company, 1989.

[9] Francisco V. Fernández. *Symbolic Analysis Techniques: application to analog design automation.* IEEE Press, New York, 1998.

[10] Francisco V. Fernández, A.Rodríguez-Vázquez, and J.L.Huertas. An advanced symbolic analyzer for the automatic generation of analog circuit design equations. In *IEEE Int. Symp. Circuits Syst.*, June 1991.

[11] Henrik Floberg. *Symbolic Analysis in Analog Integrated Circuit Design.* Kluwer Academic, New York, 1997.

[12] G.Gielen, H. Walscharts, and W.Sansen. Issac: A symbolic simulator for analog integrated circuits. *IEEE J. Solid-State Circuits.*, December 1989.

[13] G.Gielen and W.Sansen. *Symbolic Analysis for Automated Design of Analog Integrated Circuit.* Norwell, MA: Kluwer, 1991.

[14] G.Temes. Efficient method of fault simulation. In *20th Midwest Symp. Circuits Syst.*, August 1977.

[15] Marwan M. Hassoun and Kevin S. Mccarville. Symbolic analysis of large-scale networks using a hierarchical signal flowgraph approach. *Analog Integrated Circuits and Signal Processing*, March 1993.

[16] Marwan M. Hassoun and P.M.Lin. A hierarchical network approach to symbolic analysis of large-scale networks. *IEEE Transactions on Circuits and Systems -I*, April 1995.

[17] J.A.Starzyk and A. Konzykowska. Flowgraph analysis of large electronic networks. *IEEE Transactions on Circuits and Systems*, CAS-33, March 1986.

[18] Jiri and Kishore Singhal. *Computer Methods for Circuit Analysis and Design.* Van Nostrand Reinhold, New York, 1994.

[19] J.R.Abrahams and G.P.Coverley. *Signal Flow Analysis.* Oxford, New York, Pergamon Press, 1965.

[20] Jacob Katzenelson and Aaron Unikovski. Symbolic-numeric circuit analysis or symbolic circuit analysis with online approximations. *IEEE Transactions on Circuits and Systems -I*, January 1999.

[21] Charles S. Lorens. *Flowgraphs for the Modeling and Analysis of Linear Systems.* Toronto:McGraw-Hill, New York, 1964.

[22] Clifford W. Marshall. *Applied Graph Theory.* Wiley-Interscience, New York, 1971.

[23] Samuel J. Mason and Henry J. Zimmerman. *Electronic Circuits, Signals and Systems.* Wiley,New York, 1960.

[24] M.M.Hassoun. *Symbolic Analysis of Large-Scale Networks.* PhD thesis, Purdue University, 1988.

[25] M.M.Hassoun and P.M.Lin. A new network approach to symbolic simulation of large-scale networks. *IEEE, Int. Symp. Circuits Syst.*, May 1989.

[26] M.M.Hassoun and P.M.Lin. An efficient partitioning algorithm for large-scale circuits. *IEEE Int. Symp. Circuits Syst.*, pages 2405–2408, May 1990.

[27] Roy A. Nicolaides and Noel Walkington. *Maple: a comprehensive introduction.* Cambridge University Press, New York, 1996.

[28] P.M.Lin. Sensitivity analysis of large linear networks using symbolic programs. *IEEE, Int. Symp. Circuits Syst.*, May 1992.

[29] P.M.Lin and G.E.Alderson. Snap: a computer program for generating symbolic network functions. Technical report, Purdue University, August 1970.

[30] Joseph Riel. Syrup-a symbolic circuit analyzer. *Maple Tech*, 3(1), 1996.

[31] Ralph J. Smith. *Electronics: Circuits and Devices.* John Wiley & Sons, Inc., 1987.

[32] Paul W. Tuinenga. *SPICE : A Guide To Circuit Simulation And Analysis Using PSpice.* Englewood Cliffs, N.J, Toronto: Prentice Hall, 1988.

[33] Piet Wambacq, Georges G.E.Gielen, and Willy Sansen. Symbolic network analysis methods for practical analog integrated circuits: A survey. *IEEE, Transaction*

On Circuits And Systems -II, Analog And Digital Processing, 45(10), October 1998.

VITA

NAME:	Xiaofang Xie
PLACE OF BIRTH:	ShanXi, China
YEAR OF BIRTH:	1974
POST-SECONDARY EDUCATION AND DEGREES:	The University of Western Ontario London, Ontario 1999-2001 M.Sc.
	Nanjing University of Science and Technology Nanjing, P.R.China 1992-1996 B.E.
HONORS AND AWARDS:	The University of Western Ontario International Graduate Student Scholarship Special University Scholarship 1999-2001
RELATED WORK EXPERIENCE:	Teaching Assistant and Research Assistant University of Western Ontario London, Ontario 1999-2001
	Electronic Engineer Xi'an Electronic Engineering Research Institute Xi'an, P.R.China 1996-1999